Optimització i paral·lelització d'un codi de propagació d'ones elàstiques

Miguel Ferrer Ávila

25 de juny de 2010

Vull donar les gràcies als meus directors de projecte, José Ramón Herrero i Beatriz Otero, per oferir-me aquest projecte i tota la seva ajuda i paciència. A Otilio Rojas per proporcionar-nos el codi. Als meus companys de carrera, per acompanyar-me en aquest viatge.
I als meus pares i al meu germà, sense els quals no estaria avui aquí.

Índex

1

Introducció

1.1 Motivació

Avui en dia, una gran part de la feina que realitzen la majoria de persones es fa mitjançant ordinadors. Ja sigui per treballs d'investigació com per portar un registre de clients o simplement dissenyar i planificar les tasques d'una empresa, és indiscutible l'important paper que juguen els computadors a l'hora de facilitar-nos la feina. Els programes que es fan servir per dur a terme aquesta feina varien en complexitat; poden ser aplicacions senzilles, formades per un parell de línies de codi, o software de simulació molt complex que requereix una potència de càlcul addicional. És aquesta complexitat el que ens permet avaluar una aplicació pel cost que implica executar-la.

Són precisament les rutines més complexes les que ens interessen. En general, molts dels programes que realitzen càlculs científics, anàlisi numèrica, simulacions o que han de processar gran quantitat de dades comporten uns costos d'execució elevats. Aquests costos no són de caràcter econòmic (com a mínim, no directament), sinó de l'ús que fan dels recursos de la màquina on s'executen. Com a exemples de recursos, podem parlar del temps de processador, memòria, espai en disc, ample de banda de la xarxa, etc. Un ordinador estàndard pot no ser suficient per executar les aplicacions més exigents, sent necessari un supercomputador per obtenir resultats.

És, per això, interessant dissenyar els programes perquè facin un ús intel·ligent dels recursos disponibles, gestionant-los de manera adequada per treure'n el màxim profit amb el menor consum possible. El procés d'anàlisi i (re)disseny de programes per tal d'augmentar la seva eficiència s'anomena optimització. Nosaltres ens centrarem en un aspecte clau, un dels que més interessen als usu-

aris de les aplicacions: el temps d'execució.

Com a exemple d'aplicacions pràctiques podriem parlar de les simulacions en climatologia, utilitzades entre moltes altres coses per analitzar les causes i conseqüències del canvi climàtic; de meteorologia, que serveixen per predir el temps que farà; d'astronomia i astrofísica, que busquen conèixer l'estructura de l'Univers; de comportament dels mercats financiers, que permeten fer prediccions de valors i esbrinar les causes de les situacions de bonança o crisi econòmica; d'anàlisi en biologia, per tal de desxifrar genomes i seqüències de proteïnes; simulacions d'enginyeria, per tal de provar el comportament de màquines abans de la seva fabricació (avions, vaixells, cotxes de fórmula-1, etc.)

Totes aquestes àrees tan diferents tenen un denominador comú: l'ús de programes per realitzar gran quantitat de càlculs. Programes que en general tarden molt en executar-se atès la gran quantitat de dades i operacions amb les quals treballen. Una reducció significativa del temps que triguen en donar resultats permetria realitzar més simulacions en el mateix temps, i obtenir més resultats. De la mateixa manera, un programa que consumeix menys recursos de l'ordinador ens permet analitzar més dades en cada execució, oferint més resultats (o resultats més precisos).

Addicionalment, la tendència dels darrers anys a l'hora de dissenyar nous ordinadors ha consistit en replicar les unitats que conformen el xip de l'ordinador, més que millorar les ja existents. Així, amb cada nova generació de processadors augmenta el nombre de nuclis o unitats de processat, els bancs de memòria o el número de nivells de memòria cache. Sabent això, podem modificar les nostres aplicacions perquè aprofitin aquesta replicació de recursos: paral·lelitzant intel·ligentment el nostre codi aconseguirem una reducció significativa del temps d'execució i treurem més partir dels mitjans oferits per la màquina.

El nostre interès radica en posar de manifest la importància d'aquests aspectes d'optimització i paral·lelització amb un exemple real: un codi que simula la propagació d'una ona mecànica per l'escorça terrestre (ones com les produïdes amb el fregament de les plaques tectòniques). Per això, farem una anàlisi del comportament del programa a mesura que apliquem optimitzacions dirigides a reduir el temps d'execució del codi.

1.2 Objectius

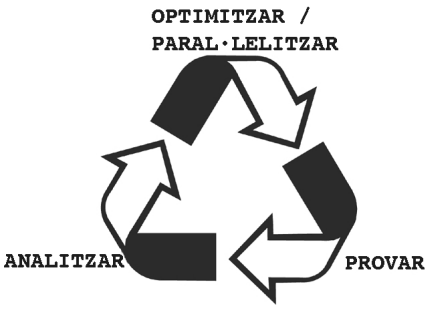
Els nostres objectius es poden resumir en tres, tots ells directament relacionats amb els altres: analitzar, optimitzar/paral·lelitzar, i avaluar.

- Analitzar: és estudiar el comportament del programa per identificar els punts conflictius on poder introduir canvis per obtenir una millora del rendiment. Detectar punts crítics on no s'aprofiten bé els recursos que la màquina ens ofereix. Aquests són els aspectes que intentem corregir. Per analitzar el programa farem servir diverses eines que el sistema operatiu (en el nostre cas, GNU/Linux) posa a la nostra disposició.

- Optimitzar i paral·lelitzar: partirem dels resultats obtinguts mitjançant l'anàlisi i el nostre coneixement de la pròpia màquina i el seu funcionament per modificar el programa, de forma que es redueixi el temps d'execució (i si és possible, altres recursos de l'ordinador). En la fase de paral·lelització, intentarem explotar el paral·lelisme de la nostra aplicació, és a dir, la capacitat de realitzar múltiples operacions a la vegada. Programarem i gestionarem l'execució simultània de diferents parts del codi amb varies APIs (*Application Programming Interface*, o interfície de programació d'aplicacions) diferents. Aquestes interfícies ens faciliten la feina, oferint-nos rutines ja creades que simplifiquen les operacions que hem de fer per obtenir el paral·lelisme.

- Avaluar: un cop hem produït una nova versió del nostre codi, hem d'avaluar el nou rendiment per detectar si hem reduït el temps d'execució i en general hem millorat l'algorisme, o si cal tornar a una versió anterior.

Seguint un esquema similar al del disseny mitjançant prototipus típic de l'Enginyeria de Software, estudiarem el programa, aplicarem optimitzacions i realitzarem mesures per avaluar si les optimitzacions es comporten com esperem i produeixen millores, o si en canvi l'impacte és negatiu i hem de retornar a una fase anterior de disseny.



1.3 Objectius específics

Formen part de l'àmbit del nostre projecte les següents tasques:

- Entendre l'esquema general de l'algorisme, què fa i com ho fa.
- Analitzar el rendiment del programa, fent servir diferents eines de mesura, per determinar els punts conflictius (on el programa passa més temps executant-se) i l'impacte de les optimitzacions sobre aquest rendiment.
- Aplicar canvis en el codi per millorar el comportament d'aquest de cara als recursos del sistema, i sobretot reduir el temps d'execució de l'aplicació, sempre sense alterar els resultats finals de l'algorisme.
- Paral·lelitzar el programa, fent servir diferents tecnologies, i comparar unes amb altres per decidir quina és la més adient.

No formen part de l'àmbit del nostre projecte les següents tasques:

- Crear un model geofísic.
- Dissenyar l'algorisme que optimitzarem.
- Modificar els resultats de l'algorisme afegint funcionalitats o alterant la lògica del model que el governa.

1.4 Metodologia de treball. La llei d'Amdahl

En aquest capítol explicarem un dels conceptes més importants a l'hora de millorar un algorisme ja dissenyat: la **llei d'Amdahl**, que ens dirà com ho hem de fer per reduir el temps d'execució d'un programa.

La llei d'Amdahl, que rep el seu nom de Gene Amdahl, un arquitecte de computadors, modela la millora màxima que podem esperar en un sistema quan introduïm una millora en una part d'aquest sistema. La llei prediu el factor de millora de velocitat (*speed-up*) que afectarà al temps general del sistema a partir del pes que la part optimitzada juga en el temps total del càlcul del sistema, i també del factor de millora d'aquesta part. Una formulació de la llei és:

$$S_T = \frac{1}{(1 - P_{opt}) + \frac{P_{opt}}{S_{P_{opt}}}} \tag{1.1}$$

on S_T és el factor de millora de velocitat del sistema general, P_{opt} és la proporció del temps total de càlcul que suposa la part optimitzada, i $S_{P_{opt}}$ és el factor de millora de temps en la part optimitzada. Una manera de definir el factor de millora és:

$$S = \frac{Temps_{original}}{Temps_{optimitzat}} \tag{1.2}$$

Per exemple, si

$$P_{opt} = 20\% = 0.2$$
$$S_{P_{opt}} = 400\% = 4\text{x}$$

llavors

$$S_T = \frac{1}{0.8 + \frac{0.2}{4}} = \frac{1}{0.805} = 1.24\text{x}$$

és a dir, si optimitzem una part del codi que suposa el 20% del temps d'execució total, i obtenim una millora del temps d'un 400% (4x, això és, el temps d'aquesta part del codi es redueix a una quarta part), la millora global de tot el sistema és d'un 24% (1.24x, tarda el 80.6% del temps original).

Provem ara un altre cas:

$$P_{opt} = 45\% = 0.45$$

$$S_{P_{opt}} = 200\% = 2\text{x}$$

llavors

$$S_T = \frac{1}{0.55 + \frac{0.45}{2}} = \frac{1}{0.775} = 1.29\text{x}$$

Si, en canvi, l'optimització obtinguda es redueix a la meitat (200% en comptes de 400%, és a dir, 2x en comptes de 4x), però s'aplica a una part més gran del codi (al 45%), el factor de millora que obtenim és més gran (1.29x, el sistema trigarà un 77% del temps original).

En general, fent proves podem veure que l'impacte que obtenim de les millores aplicades és més gran quant més gran sigui la part on l'apliquem. Unes gràfiques, extretes de [3], ens mostren la relació entre la proporció del codi optimitzat i el rendiment general del sistema:

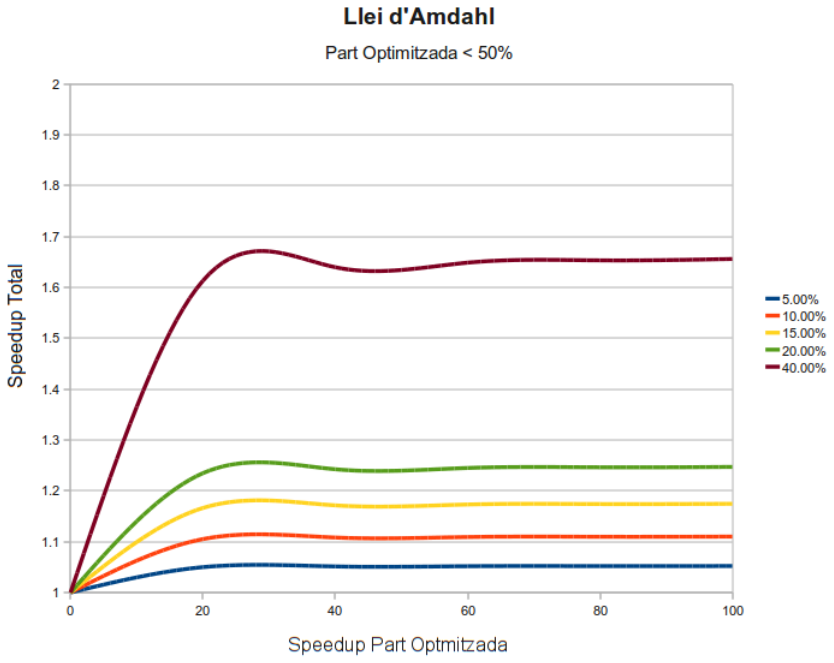


Figura 1.1: *Llei d'Amdahl, amb optimitzacions aplicades a menys del 50% del codi*

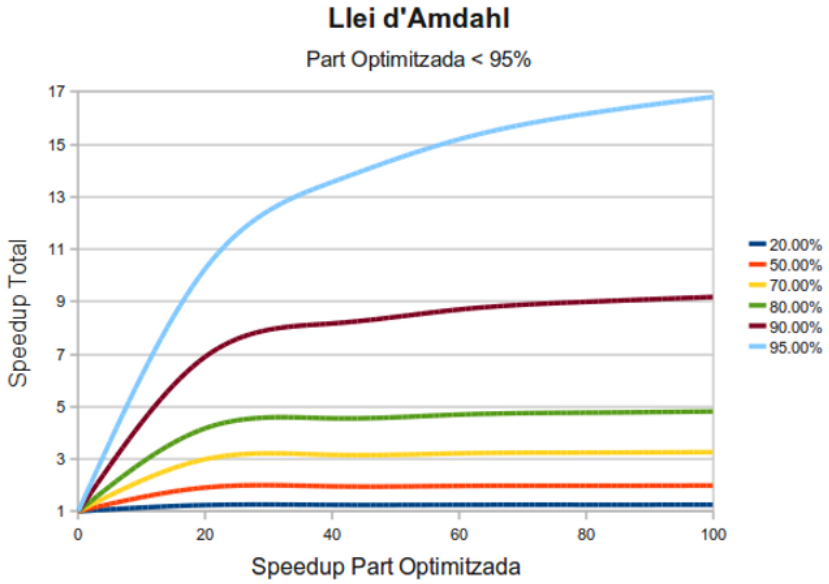


Figura 1.2: *Llei d'Amdahl, amb optimitzacions aplicades fins al 95% del codi*

En conclusió, hem de localitzar les parts del codi on el processador passa més temps realitzant operacions. Si apliquem les nostres optimitzacions en aquestes parts, l'impacte de les millores serà major en el comportament global de tot el programa.

Podem trobar més informació sobre la llei d'Amdahl a [3] i a [17].

2

El problema

El problema que ens plantejem és un codi desenvolupat pel Dr. Otilio J. Rojas, professor de la Universidad Central de Venezuela. El programa implementa un model sísmic per estudiar la propagació d'ones elàstiques per un medi material (p.ex l'escorça terrestre), representat com una superfície en dos dimensions (x i z). Aquesta superfície es discretitza en una sèrie de punts en dues malles desplaçades una respecte a l'altra (*staggered grid*), i s'estudia el desplaçament i el factor de estrès i esforç (*stress-strain*) que experimenten aquests punts quan es propaguen ones per la terra causades pel fregament de les plaques (terratrèmols).

Podem trobar una definició completa del problema i del codi en [1]

2.1 Una breu introducció a la sismologia

Les ones sísmiques són unes ones mecàniques que viatgen per les plaques tectòniques que formen l'escorça terrestre. Essencialment són com ones sonores. Quan les ones sonores viatgen per sòlids, tenen dos components bàsics: un component longitudinal i un transversal. Les ones longitudinals són aquelles en les quals les partícules que les formen oscil·len en la mateixa direcció en la que viatja l'ona. Per contra, les ones transversals són aquelles en les quals les partícules oscil·len perpendicularment respecte a la direcció en la que es desplaça l'ona.[18]

Hi han dos tipus d'ones sísmiques: les ones internes i les ones superficials. En cadascun d'ells tipus trobem dues ones diferents.

Les **ones internes** són aquelles que viatgen per l'interior de l'escorça terrestre:

- Ones-P: són ones longitudinals o de compressió. Poden desplaçar-se per qualsevol medi. Quan ho fan per l'aire formen so.
- Ones-S: són ones transversals, on el terra es desplaça perpendicularment respecte al moviment de l'ona. Només poden desplaçar-se per sòlids, i ho fan molt més lentament que les ones-P.

Les **ones superficials** són les ones que viatgen per la superficie terrestre. Poden ser les més destructives atès la seva gran amplitud i duració:

- Ona de Rayleigh: són ones que es comporten com les que es formen a la superfície de l'aigua quan cau un objecte dintre.
- Ona de Love: ones que produeixen un efecte de cisalla circular sobre el terra.

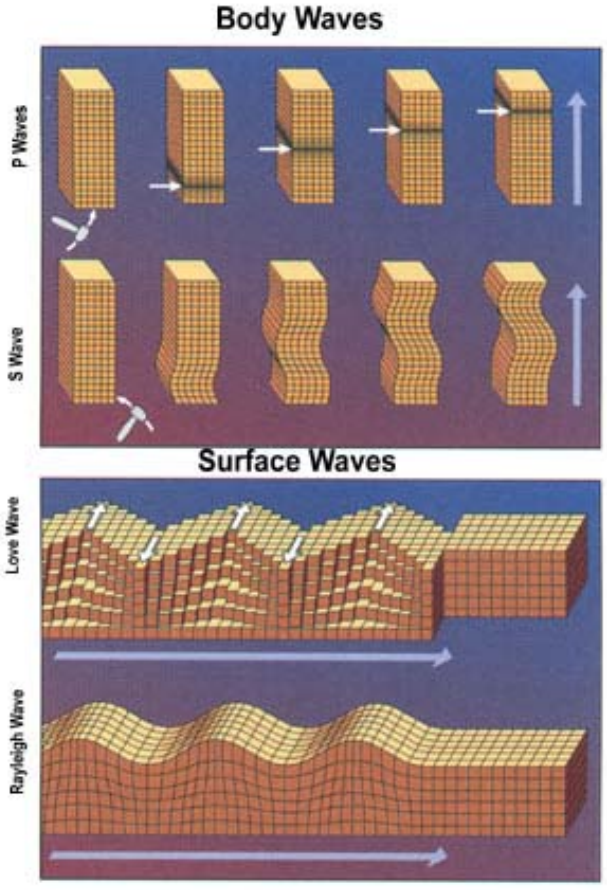


Figura 2.1: *Les components de les ones sísmiques*

El nostre algorisme treballa amb les ones P, S i el pols Rayleigh en un medi homogeni pel qual es coneix una solució analítica amb la qual podem comparar els nostres resultats. El problema rep el nom de P-SV, per la presència d'ones P i ones SV. Les ones SV són ones S que estan polaritzades en un pla vertical, això és, la direcció de l'oscil·lació de las partícules que es desplacen a causa de l'ona és vertical.

2.2 El model analític

Comencem considerant un semiplà elàstic infinit i homogeni que representa una secció vertical de l'escorça i mantell terrestre, amb l'eix x horitzonal i l'eix z vertical i positiu cap avall (cap a l'interior de la Terra). La propagació de les ones elàstiques ve descrit per les equacions:

$$\rho\frac{\partial^2 u}{\partial t^2} = \frac{\partial}{\partial x}\left[\tau_{xx}\right] + \frac{\partial}{\partial z}\left[\tau_{xz}\right] \tag{2.1}$$

$$\rho\frac{\partial^2 w}{\partial t^2} = \frac{\partial}{\partial x}\left[\tau_{xz}\right] + \frac{\partial}{\partial z}\left[\tau_{zz}\right] \tag{2.2}$$

$$\tau_{xx} = (\lambda + 2\mu)\frac{\partial u}{\partial x} + \lambda\frac{\partial w}{\partial z} \tag{2.3}$$

$$\tau_{zz} = (\lambda + 2\mu)\frac{\partial w}{\partial z} + \lambda\frac{\partial u}{\partial x} \tag{2.4}$$

$$\tau_{xz} = \mu\left(\frac{\partial u}{\partial z} + \frac{\partial w}{\partial x}\right) \tag{2.5}$$

on u i w representen el vector de desplaçament de les partícules, (u, w), i τ_{xx},τ_{xz} i τ_{zz} les components del tensor de estrès o tensor d'esforç. μ, λ i ρ són paràmetres calculables mitjançant fórmules senzilles. $\rho(x, z)$ és la densitat del sistema, i $\mu(x, y)$ i $\lambda(x, y)$ són els denominats paràmetres de Lamé, que serveixen per satisfer la llei de Hooke (la llei fonamental dels moviments oscil·latoris i de les ones). A partir d'aquests paràmetres, es poden calcular les velocitats de les ones P (α) i S (β) com:

$$\alpha = \sqrt{\frac{\lambda + 2\mu}{\rho}} \tag{2.6}$$

$$\beta = \sqrt{\frac{\mu}{\rho}} \tag{2.7}$$

Aquestes equacions governen el comportament de les ones en l'interior del pla. A la superfície ($z = 0$) considerem una situació particular. Allà acaba el medi homogeni, i no hi ha fregament de les partícules amb cap altre medi. Per això, en la superfície les components normal i tangencial dels tensors d'esforç es fan

zero i tenim el que s'anomena una superfície lliure.

$$z = 0 \Rightarrow \tau_{xz} = \tau_{zz} = 0 \tag{2.8}$$

La situació es pot comparar a moure un cordill, els extrems del qual no estan lligats a res, des d'un dels seus extrems amb un moviment oscil·latori, i veure com es comporta l'altre extrem lliure.

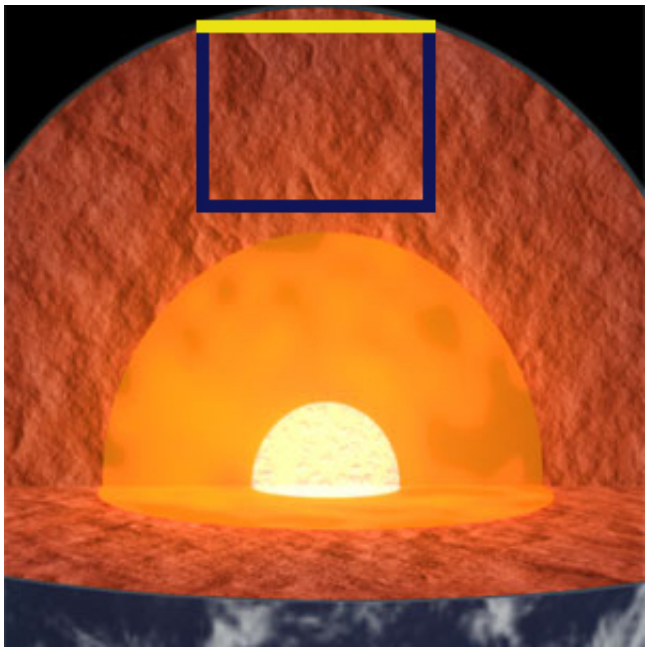


Figura 2.2: *Secció terrestre on es pot veure un exemple del pla amb el qual treballem. En groc, hem marcat la superfície lliure*

Finalment, per completar el model, s'inclou un punt del pla on s'inicia una explosió que posa en moviment el sistema. El pols és de tipus gaussià perquè té forma de funció gaussiana:

$$f(t) = \exp(-\delta(t - t_0)^2) \tag{2.9}$$

2.3 Resolució del problema

En aquesta secció discutirem com resol el model analític de l'apartat anterior el programa que hem d'optimitzar. En un primer apartat explicarem què vol calcular el codi. Seguidament introduirem dos conceptes teòrics que són claus per la resolució del problema: el teorema de Taylor i el teorema del valor mitjà. Finalment, explicarem com aplica el programa aquests dos conceptes teòrics, i presentarem una versió esquemàtica en pseudocodi de l'aplicació.

2.3.1 Resolució del model analític

Què calcula el programa? Com ja hem dit, el codi simula la propagació d'una ona elàstica per un medi material, complint les equacions del model citades. El medi es representa mitjançant un semiplà, el qual discretitzem en una sèrie de punts que conformen una malla. Cadascun d'aquests punts o partícules té associat un moviment produït per la propagació de l'ona. El programa reprodueix com es mouen aquestes partícules al llarg del temps. Per tant, el que ens interessa és el vector de desplaçament (u, w) que té associat cada partícula, en un moment determinat (t).

En les equacions del model analític de l'apartat 2.2 apareix un terme que, tot i no ser geofísics, ens és prou conegut: l'acceleració. En la part esquerra de les fórmules 2.1 i 2.2 podem veure la segona derivada del vector de desplaçament de les partícules respecte al temps (és a dir, l'acceleració de les partícules):

$$\rho \frac{\partial^2 u}{\partial t^2} = \rho \ddot{u} = \rho a_u \tag{2.10}$$

$$\rho \frac{\partial^2 w}{\partial t^2} = \rho \ddot{w} = \rho a_w \tag{2.11}$$

Aquesta acceleració és la que experimenten les diferents partícules a causa de la propagació de les ones. A partir de la definició de la derivada (derivada parcial, en aquest cas) d'una funció lineal, podem deduir la fórmula aproximada de com canvia el vector desplaçament amb el temps:

$$\frac{\partial f}{\partial t}(t_0) = \lim_{\Delta t \to 0} \frac{f(t_0 + \Delta t) - f(t_0)}{\Delta t}$$

Si ara repetim la fórmula de la derivada per obtenir la segona derivada, amb Δt molt petita, i despreciant l'error:

$$\frac{\partial^2 f}{\partial t^2}(t_0) \approx \frac{\frac{f(t_0+2\Delta t)-f(t_0+\Delta t)}{\Delta t}}{\Delta t} - \frac{\frac{f(t_0+\Delta t)-f(t_0)}{\Delta t}}{\Delta t}$$

Per facilitat la lectura, farem les següents substitucions:

$$f(t_0) \equiv f_{old}$$
$$f(t_0 + \Delta t) \equiv f$$
$$f(t_0 + 2\Delta t) \equiv f_{new}$$

Així, la fórmula anterior ens queda:

$$\frac{\partial^2 f}{\partial t^2}(t_0) \approx \frac{\frac{f_{new}-f}{\Delta t}}{\Delta t} - \frac{\frac{f-f_{old}}{\Delta t}}{\Delta t}$$

$$\frac{\partial^2 f}{\partial t^2}(t_0) \approx \frac{f_{new}-f}{\Delta t^2} - \frac{f-f_{old}}{\Delta t^2}$$

i, agrupant els termes:

$$\frac{\partial^2 f}{\partial t^2}(t_0) \approx \frac{f_{new}-2f+f_{old}}{\Delta t^2}$$

Si ara substituïm a les equacions 2.1 i 2.2, fent $f = U$ i $f = W$:

$$\rho\frac{U_{new}-2U+U_{old}}{\Delta t^2} \approx \frac{\partial}{\partial x}\left[\tau_{xx}\right] + \frac{\partial}{\partial z}\left[\tau_{xz}\right]$$
$$\rho\frac{W_{new}-2W+W_{old}}{\Delta t^2} \approx \frac{\partial}{\partial x}\left[\tau_{xz}\right] + \frac{\partial}{\partial z}\left[\tau_{zz}\right]$$

I finalment,

$$U_{new} = 2U - U_{old} + \frac{\rho}{\Delta t^2}\left(\frac{\partial \tau_{xx}}{\partial x} + \frac{\partial \tau_{xz}}{\partial z}\right) \tag{2.12}$$

$$W_{new} = 2W - W_{old} + \frac{\rho}{\Delta t^2}\left(\frac{\partial \tau_{xz}}{\partial x} + \frac{\partial \tau_{zz}}{\partial z}\right) \tag{2.13}$$

Juntament amb les equacions dels tensors, 2.3, 2.4 i 2.5, aquestes fórmules ens donen la solució a qué ha de calcular l'algorisme per obtenir els vectors de desplaçament, U i W, en qualsevol moment:

$$\tau_{xx} = (\lambda + 2\mu)\frac{\partial u}{\partial x} + \lambda \frac{\partial w}{\partial z}$$

$$\tau_{zz} = (\lambda + 2\mu)\frac{\partial w}{\partial z} + \lambda \frac{\partial u}{\partial x}$$

$$\tau_{xz} = \mu\left(\frac{\partial u}{\partial z} + \frac{\partial w}{\partial x}\right)$$

$$U_{new} = 2U - U_{old} + \frac{\rho}{\Delta t^2}\left(\frac{\partial \tau_{xx}}{\partial x} + \frac{\partial \tau_{xz}}{\partial z}\right)$$

$$W_{new} = 2W - W_{old} + \frac{\rho}{\Delta t^2}\left(\frac{\partial \tau_{xz}}{\partial x} + \frac{\partial \tau_{zz}}{\partial z}\right)$$

Primer, obtenim els tensors derivant en les diferents direccions els vectors de desplaçament i multiplicant-los pels paràmetres λ i μ. Un cop hem calculat els tensors, els derivem respecte a les direccions que ens interessen i els multipliquem per ρ per obtenir els nous vectors de desplaçament U_{new} i W_{new} i tornem a començar.

2.3.2 Teorema i desenvolupament de Taylor per a una funció. Teorema del valor mitjà

Per entendre com solucionar el problema anterior amb l'algorisme proposat, és important conèixer abans el teorema de Taylor i el teorema del valor mitjà (que és un cas particular d'un altre teorema, el de Rolle, i que s'utilitza per demostrar el teorema de Taylor).

Situem-nos en un espai $\mathbb{R}^n$ i considerem una bola B (un conjunt de punts compresos entre un centre i una distancia màxima en qualsevol direcció, anomenada radi de la bola) centrada en un punt a amb $a \in \mathbb{R}^n$. Sigui f una funció definida en $\bar{B}$ (clausura de B, un conjunt tancat que conté els punts de B i els seus punts d'acumulació) amb $n+1$ derivades parcials continues (condició suficient

de diferenciabilitat, és a dir, que f és diferenciable en $\bar{B}$). Llavors, es pot definir f com:

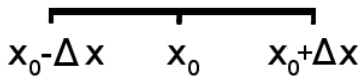
$$f(x) = \sum_{\alpha=0}^{\alpha=n} \frac{1}{\alpha!} \frac{\partial^\alpha f(a)}{\partial x^\alpha} (x-a)^\alpha + \sum_{\alpha=n+1} R(x)(x-a)^\alpha \qquad (2.14)$$

És a dir, que la funció es pot descompondre en la suma de productes de les derivades parcials, d'ordre entre 0 i n, i un residu, d'ordre superior (que a la pràctica es descarta o no es considera). Aquesta suma rep el nom de desenvolupament de Taylor de la funció f en l'entorn d'a. Aquest desenvolupament és molt important perquè ens permet aproximar una funció com un polinomi d'un cert ordre, menyspreant el residu o error. Quant més termes de la sèrie agafem, amb més precisió ens aproximem a la funció considerada. Per facilitar la lectura, i perquè a efectes d'aquest document és suficient, farem servir l'expressió del teorema pel cas d'una única variable:

$$f(x) = \sum_{\alpha=0}^{\alpha=n} \frac{f^{(\alpha)}(a)}{\alpha!} (x-a)^\alpha + \sum_{\alpha=n+1} R(x)(x-a)^\alpha \qquad (2.15)$$

on $f^{(\alpha)}(a)$ representa la derivada α-éssima de f, i a és un punt del domini de f entorn al qual aproximem la funció.

El teorema del valor mitjà s'utilitza per demostrar el teorema de Taylor, i ens apareix si manipulem expressions d'aquest. Per facilitat la representació, l'explicarem amb una funció d'una única variable.
Sigui f una funció continua i derivable en un entorn de x_0. Suposem que coneixem el valor de la funció en dos punts de l'entorn, per exemple, $f_{left} = x_0 - \Delta x$ i $f_{right} = x_0 + \Delta x$ i volem saber el valor aproximat de la derivada de f en aquest entorn.



Mitjançant el desenvolupament de Taylor, sabem que:

$$f(x) = f(x_0) + \frac{\partial f(x_0)}{\partial x}(x - x_0) + \frac{\partial^2 f(x_0)}{\partial x^2}\frac{(x - x_0)^2}{2} + ...$$

Ens quedem amb ordre 2 i descartem el residu. Evidentment, per a $x = x_0$, $f(x) = f(x_0)$. Per a $x = x_0 - \Delta x$, tenim:

$$f_{left} = f(x_0 - \Delta x) \approx f(x_0) + \Delta x \frac{\partial f(x_0)}{\partial x} + (\Delta x)^2 \frac{\partial^2 f(x_0)}{\partial x^2}$$

i per a $x = x_0 + \Delta x$:

$$f_{right} = f(x_0 + \Delta x) \approx f(x_0) - \Delta x \frac{\partial f(x_0)}{\partial x} + (\Delta x)^2 \frac{\partial^2 f(x_0)}{\partial x^2}$$

Si ara els restem, $f_{right} - f_{left}$, ens queda:

$$f_{right} - f_{left} \approx -2\Delta x \frac{\partial f(x_0)}{\partial x}$$

Aïllant la derivada, que és el que ens interessarà aproximar, tenim que:

$$\frac{\partial f(x_0)}{\partial x} \approx \frac{f_{left} - f_{right}}{2\Delta x} \tag{2.16}$$

És a dir, que coneguts els valors pròxims, es pot deduir el valor aproximat de la derivada en un punt mig. Aquesta tècnica rep el nom d'aproximació a la derivada per diferencia finita central (*centered difference approximation*) [2] i és la que dóna lloc a les operacions de *stencil* o plantilla que veurem en el següent apartat.

En aquest apartat hem explicat breument el teorema de Taylor i el teorema del valor mitjà, que ens interessa perquè el farem servir en l'algorisme per facilitar el càlcul de derivades parcials. Hi ha molt més a dir respecte al teorema de Taylor, però aquest tema s'allunya de l'àmbit del projecte. El que és important és saber que les funcions que satisfan els requisits esmentats abans es poden aproximar amb el desenvolupament de Taylor, i l'algorisme del problema aprofita aquesta qualitat per realitzar el càlcul de derivades.

2.3.3 L'algorisme H-MSSG

Un cop deduïdes les fórmules que calcula el nostre codi, ens hem de preguntar: com les resolem? Algunes de les operacions involucrades són tan senzilles com sumes, restes i multiplicacions. Però, què hi ha de la derivació parcial? És en aquest punt quan intervenen els conceptes que hem introduït del desenvolupament de Taylor, el teorema del valor mitjà i l'aproximació a la derivada per diferència finita central.

Com ja hem dit quan parlàvem del model, el semipla que constitueix el medi on es propaga l'ona es discretitza, formant una malla de partícules. No obstant, per implementar de forma eficient els vectors de desplaçament i els tensors de cada punt, el programa utilitza un concepte anomenat *Standard Staggered Grid* (SSG) o malla estàndard desplaçada. La idea darrera de la malla desplaçada té molt a veure amb el teorema del valor mitjà per aproximar el valor de les derivades respecte a direccions de l'espai.

La informació que ens interessa guardar en el nostre sistema és:

- Els vectors de desplaçament de tots els punts o partícules, U i W.
- Els tensors en cada punt, $\tau_{xx}, \tau_{xz}, \tau_{zz}$.

Per mantenir aquesta informació establim dues malles sobre el domini. Aquestes dues malles estan desplaçades una respecte l'altra, de forma que els nusos d'una de les malles es troben al centre dels quadrats formats per l'altra malla:

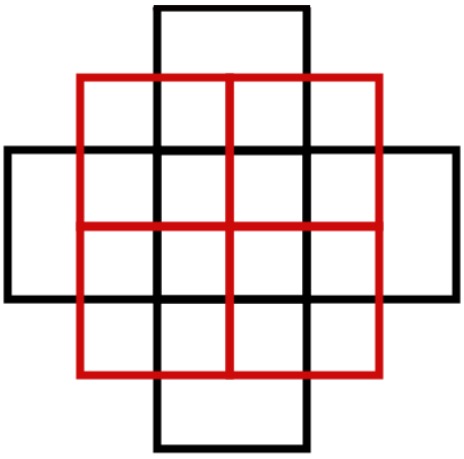
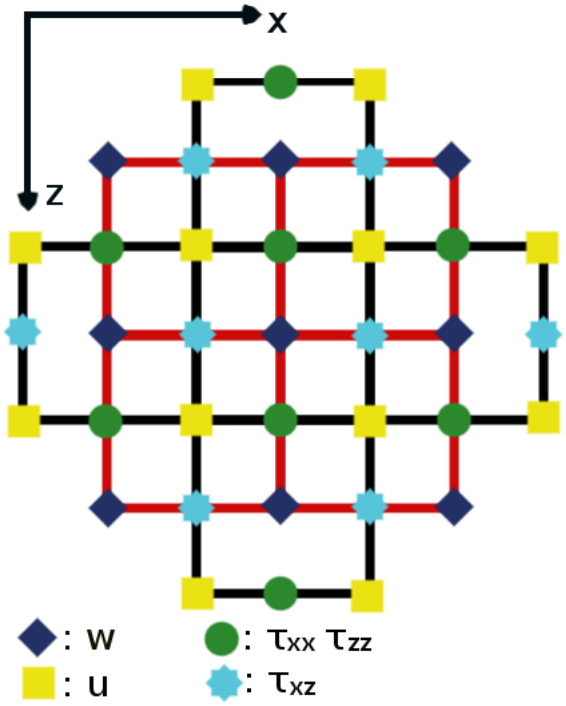


Figura 2.3: *Dues malles desplaçades*

Els nusos d'una de les malles representen els valors de u per les diferents partícules, mentre que els de l'altra malla representen els valors de w. La distància entre els nusos depèn de la mida de la secció de pla que considerem i del nombre de punts amb els que el discretitzem: $\Delta x = \frac{long_x}{num_x}$ i $\Delta z = \frac{long_z}{num_z}$. Els punts d'intersecció de les malles representen valors dels tensors (que, recordem, s'obtenen derivant u i w respecte a les seves direccions). Així, gràcies a l'aproximació de les derivades de l'apartat anterior i aquesta representació amb malles desplaçades, és fàcil calcular i localitzar els valors dels tensors a partir dels valors adjacents d'u i w.

En el següent esquema es poden veure els valors col·locats sobre la malla desplaçada. Com s'obté un valor depèn de la posició on es trobi: per exemple, si partim d'un nus $u_{i,j}$ i avancem horitzontalment, trobarem el tensor τ_{xx} i el tensor τ_{zz} que s'obtenen derivant $u_{i,j}$ i $u_{i+\Delta x,j}$ respecte de x (i derivant w respecte de z, com es pot veure ja que arribem al mateix nus si recorrem verticalment la malla des de $w_{i,j}$ i $w_{i,j+\Delta z}$). Anàlogament, si partim d'un nus $u_{i,j}$ i avancem verticalment, trobarem un tensor τ_{xz} que s'obté derivant u respecte de z i w respecte de x.
La distància entre dos valors adjacents és sempre $\frac{\Delta z}{2}$ en direcció vertical i $\frac{\Delta x}{2}$ en direcció horitzontal.



En realitat, els valors dels tensors per cada partícula s'aproximen mitjançant 4 valors adjacents d'u i w (aproximació de quart ordre), per ser més precisos. Similarment, els valors d'u_{new} i w_{new} es calculen a partir de 4 valors adjacents dels diferents tensors. Aquests valors s'agafen tant de l'esquerra com de la dreta del valor que volem calcular. En els límits laterals, l'aproximació és de major ordre (fins a ordre 6), però només cap al costat on disposem de valors. I recordem que sobre la superfície lliure ($z = 0$) els tensors τ_{xz} i τ_{zz} s'anul·len, i són un cas particular.

Aquest mètode de distribució de valors en nodes adjacents a un punt d'interès mitjançant un aproximació numèrica determinada s'anomena plantilla (en anglès, *stencil*), i és molt utilitzada en el món de la computació científica, principalment (i com és el nostre cas) per resoldre derivades parcials. El mètode de plantilla serà un dels principals limitadors a les optimitzacions i al paral·lelisme que ens trobarem en el programa, perquè genera moltes **dependències de dades**, ja que per calcular qualsevol valor en un punt de la malla necessitarem disposar d'uns quants valors adjacents.

El nom de l'algorisme és H-MSSG. La H (*Horizontal*) indica que la superfície lliure es situa en la mateixa direcció del desplaçament horitzontal (x). La M *Mimetic* indica que està basat en un mètode mimètic: això és, que realitza operacions en un entorn discret simulant un continu real. I SSG indica que treballa amb una *Standard Staggered Grid*, com acabem d'explicar.

2.4 Pseudocodi del programa

El codi original que realitza la simulació explicada en l'apartat anterior, i el qual ens hem proposat optimitzar, està escrit en Fortran i ocupa unes 2000 línies de codi font. Per facilitar la comprensió del programa, presentem en aquesta secció una versió resumida i en pseudocodi.

És important destacar que és una versió esquemàtica, per explicar com funciona, i hi ha unes quantes diferències amb el codi original.

```
var
  GLOBAL real L, L2M, M, R, chi1;
end
psv_ssg2_homog
begin program
  var
    matriu Uold, U, Unew;
    matriu Wold, W, Wnew;
    matriu Txx,Txz, Tzz;
    enter i, nombreIteracions;
    real time, dt;
  end
  call lectura_i_inicialitzacio_dades(Unew, U, Uold, Wnew, W, Wold, Txx, Txz, Tzz, dt);
  call nombre_iteracions(nombreIteracions); //nombreIteracions és múltiple de 3
  time := 0;
  while i ≤ nombreIteracions do
        ----------------------------------------
        //Uold → U → Unew
        call calcula_Tensors(Txx, Txz, Tzz, U, W);
        call calcula_U_W(Unew, Wnew, Txx, Txz, Tzz, U, W, Uold, Wold);
        call calcula_Superficie_Lliure(Unew, Wnew, U, Uold, Txx, Txz, Tzz);
        time := time + dt;
        ----------------------------------------
        //U → Unew → Uold
        call calcula_Tensors(Txx, Txz, Tzz, Unew, Wnew);
        call calcula_U_W(Uold, Wold, Txx, Txz, Tzz, Unew, Wnew, U, W);
        call calcula_Superficie_Lliure(Uold, Wold, Unew, U, Txx, Txz, Tzz);
        time := time + dt;
        ----------------------------------------
        //Unew → Uold → U
```

```
        call calcula_Tensors(Txx, Txz, Tzz, Uold, Wold);
        call calcula_U_W(U, W, Txx, Txz, Tzz, Uold, Wold, Unew, Wnew);
        call calcula_Superficie_Lliure(U, W, Uold, Unew, Txx, Txz, Tzz);
        time := time + dt;
        - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
        i := i + 1;
    endwhile
    end program


proc calcula_Tensors(Txx, Txz, Tzz, U, W)
    begin
      var
        INOUT matriu U,W;
        INOUT matriu Txx,Txz, Tzz;
        matriu DerX, DerZ;
      end
      // Derivem U respecte x i W respecte z
      call UxWz(U, W, DerX, DerZ);
      Txx := L2M * DerX + L * DerZ;
      Tzz := L2M * DerZ + L * DerX;
      - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
      // Derivem W respecte x i U respecte z
      call UzWx(U, W, DerX, DerZ);
      Txz := M * (DerZ + DerX);
    end proc calcula_Tensors


proc calcula_U_W(Unew, Wnew, Txx, Txz, Tzz, U, W, Uold, Wold)
    begin
      var
        INOUT matriu Unew, U, Uold;
        INOUT matriu Wnew, W, Wold;
        INOUT matriu Txx,Txz, Tzz;
        matriu DerX, DerZ;
      end
      Unew := 2 * U - Uold;
      Wnew := 2 * W - Wold;
      // Derivem Txx respecte x i Txz respecte z
      call DxTxx_DzTxz(Txx, Txz, DerX, DerZ);
```

```
    Unew := Unew + R * (DerX + DerZ);
    - - - - - - - - - - - - - - -
    // Derivem Txz respecte x i Tzz respecte z
    call DxTxz_DzTzz(Txz, Tzz, DerX, DerZ);
    Wnew := Wnew + R * (DerX + DerZ);
  end proc calcula_U_W


proc calcula_Superficie_Lliure(Unew, Wnew, U, Uold, Txx, Txz, Tzz)
  begin
    var
      INOUT matriu Unew, U, Uold;
      INOUT matriu Wnew;
      INOUT matriu Txx,Txz, Tzz;
      real Ux, dxTxx, dzTxz, i;
    end
    for i := 1 to nx do
        Unew[0][i] := 2 * U[0][i] - Uold[0][i];
        call deriv(dxTxx, dzTxz, i);
        Unew[0][i] := Unew[0][i] + R * (dxTxx + dzTxz);
    od
    for i := 1 to nx do
        Ux := grad(Unew[0][i]);
        call deriv(dxTxx, dzTxz, i);
        Wnew[0][i] := grad(W[0][i]) - chi1 * Ux;
    od
  end proc calcula_Superficie_Lliure
```

Com es pot veure, a l'hora de treballar amb les malles del problema, aquestes es tradueixen en matrius de valors reals (una posició per partícula de la malla). Tenim una matriu U per la component u del vector de desplaçament de cada partícula, una matriu W per la component w, i una matriu pel valor dels tensors en cada punt.

Les matrius U, $Unew$ i $Uold$ representen diferents estats en el temps dels valors de u per totes les partícules. El codi comença amb l'estat actual a U, i a mesura que avança el temps en passos de durada dt, es guarda el valor a $Unew$, després a $Uold$ i torna a guardar-se a U, $3 * dt$ segons després. Tot això es fa de forma anàloga per a W.

3

Arquitectura de l'experiment i tecnologia

El propòsit d'aquesta part és explicar com hem dissenyat l'experiment i la tecnologia emprada per fer les mesures.

3.1 Els experiments

El comportament del nostre programa es pot regular mitjançant els fitxers d'entrada de dades. En aquest fitxers es poden definir els valors que prendran diferents paràmetres de l'experiment, i d'aquesta forma, modificar aspectes com l'amplitud de la secció del medi considerat, la quantitat de punts de les malles, o el temps total de la mesura de l'ona.

La major part del valor dels paràmetres s'especifiquen al fitxer "datain". Algunes variables reben el valor del fitxer "grad_parameter", però aquestes són més tècniques, i per donar valors acceptables cal entendre la física del programa. Per això, nosaltres ens centrarem en donar valors amb el fitxer datain.

Per mostrar l'impacte de les optimitzacions aplicades al codi, treballem amb 3 fitxers datain, que formen malles de diferent mida. La malla petita és de 899x899 punts. La malla de mida mitjana és de 1399x1399 punts. La malla gran és de 2299x2299 punts. Com podem comprovar als experiments, a major mida de malla (major nombre de partícules), major mida tenen les matrius internes del programa. Això fa que s'hagin de fer més càlculs, i el programa tarda més. Per veure l'efecte les optimitzacions, executem cadascuna de les versions del nostre programa amb els tres datain de diferents mides, i mesurarem els seus temps

d'execució.

Addicionalment, fem servir un datain que genera una malla molt més gran i simula la propagació durant molt més temps. Aquesta malla és de 4819x4819 elements. Atès la mida del problema, aquesta simulació tarda hores en acabar amb el codi sense optimitzar. Per aquest motiu, només l'executem amb el programa original i la millor versió un cop hem aplicat totes les optimitzacions i l'hem paral·lelitzada.

Adjuntem una còpia dels fitxers amb les dades d'entrada en l'annex A.

3.2 Tecnologia i característiques de les màquines

Per fer les simulacions i provar les optimitzacions, fem servir dos ordinadors amb especificacions diferents. En aquest apartat llistarem les seves característiques tècniques més rellevants, les quals afectaran de forma més o menys directa al rendiment de l'aplicació. També inclourem un breu resum de les diferents aplicacions i interfícies que hem fet servir al llarg de tot el procés d'optimització, com per exemple, els dos compiladors que hem usat per generar el programa. Aquest dos compiladors són força diferents, i veurem que això també impacta de forma directa en el comportament del programa.

3.2.1 Les màquines

La primera màquina és un servidor, situat a la FIB, i és a la màquina a la qual enfoquem les optimitzacions, principalment perquè, al tenir més nuclis i més nivells de memòria cache, és la que ofereix més potencial per optimitzar el programa. En general, la major part d'optimitzacions també tenen un efecte positiu en l'altra màquina considerada, però hi ha alguns centrats en explotar específicament les característiques del servidor.

El servidor té com a processador un Intel Core i7, amb arquitectura Nehalem. Les característiques més destacades són:

- 2 processadors *quad-core*, que sumen en total 8 nuclis amb una freqüència de 2.27 GHz.
- Els 8 nuclis ens donen una capacitat d'execució de 8 threads en paral·lel.
- Un primer nivell de memòria cache (L1) de 32 KB d'instruccions i 32 KB de dades, per cada nucli.
- Un segon nivell de cache (L2), de 256 KB, per cada nucli.
- Un tercer nivell de cache (L3), de 8 MB, compartit entre tots els nuclis.
- Instruccions vectorials: fins a SSE 4.2

A més, disposem de 8 GB de memòria RAM.

L'altre ordinador és un portàtil amb un processador Intel Core2 Duo Penryn 9600, amb les següents característiques:

- Un processador amb dos nuclis de 2.66 GHz de freqüència.
- Els dos nuclis ens donen una capacitat d'execució de 2 threads en paral·lel.
- Un primer nivell de cache (L1) amb 32 KB per dades i 32 KB per instruccions, per cada nucli.
- Un segon nivell de cache (L2), compartida entre els dos nuclis, amb 6 MB.
- Instruccions vectorials: fins a SSE 4.1

En quant a memòria, aquesta màquina disposa de 4 GB de RAM.

En ambdós ordinadors, el sistema operatiu és GNU/Linux (Ubuntu).

Per comoditat, les optimitzacions (aquelles que no són específiques de l'arquitectura del servidor) es proven primer en el portàtil, i després són enviades al servidor per avaluar-les amb aquella màquina.

3.2.2 Eines de mesura

Les eines de mesura disponibles al sistema operatiu GNU/Linux són molt adequades per analitzar el programa i el seu comportament. Les que fem servir són:

- **GNU time**: aquesta comanda ens mostra el temps d'execució total de l'aplicació, així com el temps que passa executant codi en entorn d'usuari i en entorn de sistema. També ens ofereix un percentatge d'ús de CPU, especialment útil quan paral·lelitzem el codi. No cal recompilar el programa, i per això és molt ràpid fer mesures amb ella.
  No obstant, a vegades li manca precisió, per la qual cosa farem servir també altres eines de *profiling*. S'ha de vigilar especialment que el percentatge d'ocupació de la CPU per part de l'aplicació sigui suficient, perquè és una eina de mesura global, i no identifica quan altres processos interfereixen en la mesura.

  Està disponible en la majoria de sistemes GNU/Linux.

- **gprof**: l'eina bàsica d'anàlisi de GNU. Compilant el codi amb opció de debug i inserint automàticament codi d'instrumentació, el nostre executable generarà un fitxer amb estadístiques de rendiment (principalment, nombre

de vegades que es crida a una subrutina i temps que passa executant-se aquesta). Ofereix resultats molt intuïtius. Dona informació de temps limitada de les biblioteques (que varia si són estàtiques o dinàmiques), i no ofereix dades de temps de sistema operatiu.

Com l'anterior comanda, aquesta també està disponible en la majoria de sistemes GNU/Linux.

- **Oprofile**:l'Oprofile és una eina molt potent per realitzar mesures, ja que fa servir molts comptadors interns del processador per donar-nos informació de l'execució d'un programa. Ens pot oferir informació sobre la quantitat d'encerts i errors a la memòria cache, el nombre de cicles que un processador està ocupat amb operacions, errors que ha comés amb la predicció de salts, etc. Ens pot oferir els resultats amb una granularitat molt fina, a nivell d'instrucció o línia de codi font. A causa del format dels resultats (anotacions al costat del codi font), no posem resultats d'Oprofile de forma específica en el document.

  Es pot trobar als repositoris de Linux, o també a `http://oprofile.sourceforge.net/download/`.

- **objdump**: aquesta eina ens permet desensamblar fitxers objecte per veure el codi. Concretament, nosaltres el farem servir per saber si els compiladors introdueixen instruccions vectorials automàticament.

  Inclosa en el sistema GNU/Linux.

- **ltrace**: ltrace ens mostra el nombre de crides a les llibreries dinàmiques que fa el nostre programa, i el temps d'execució de cada tipus de crida.

  També inclosa en el sistema operatiu GNU/Linux.

3.2.3 Entorns de desenvolupament

A més de les eines de mesura, fem servir una altra aplicació, un entorn de desenvolupament per programar, que ens facilita la feina detectant errors d'escriptura de manera immediata en l'editor que té integrat. El que seleccionem per aquest projecte és Eclipse, dissenyat per IBM i l'Eclipse Foundation. És gratuït, lliure i té la seva pròpia llicencia software, la Eclipse Public License. Seleccionem aquest entorn perquè dona suport tant a codi Fortran com a C/C++, i és fàcil de fer servir.

Podem trobar aquest entorn de desenvolupament a http://www.eclipse.org/downloads/.

3.2.4 Compiladors

Per generar el codi màquina que es pugui executar, hem de fer servir un compilador per al llenguatge amb el qual s'ha escrit l'aplicació.
Inicialment, hem optar per **gfortran**, ja que forma part de la GNU Compiler Collection (GCC) i és gratuït i està ben documentat. A més, ofereix opcions que ens són necessàries, com per exemple suport a la tecnologia OpenMP (de la qual en parlarem més endavant). Malgrat això, en arribar a un punt determinat del procés d'optimització, la vectorització automàtica del codi, gfortran s'ha mostrar limitat en aquest sentit i per això hem decidit fer servir el compilador d'Intel, **ifort**.

Aquest altre compilador és software propietari i no és gratuït: les seves llicencies oscil·len entre $600 i $1400. Afortunadament, ofereixen una versió de prova per un mes (i renovable si tornem a instal·lar la versió de prova) que ens ha permès provar les optimitzacions a fons. Per defecte, el compilador d'Intel introdueix una gran quantitat d'optimitzacions en el nostre codi. Com que els processadors que fem servir han estat dissenyats per la mateixa companyia, el compilador d'Intel aplica més optimitzacions que el compilador de GNU, i per això en general els temps d'execució són menors amb ifort.

Tanmateix, hi ha una optimització per defecte del compilador d'Intel que no acceptem: l'arrodoniment dels càlculs amb reals. Per poder garantir que les optimitzacions que apliquem no tenen errors i ofereixen els mateixos resultats que el codi original sense optimitzar, ens és necessari poder comparar els resultats de la versió del programa sense optimitzar amb els resultats del codi optimitzat. Per això, demanarem al compilador d'Intel que no faci arrodoniment dels resultats de les operacions amb reals (amb la opció '-fp-model precise').

Gfortran i gcc estan disponibles als repositoris de GNU/Linux. Les versions d'avaluació de ifort i icc es poden trobar a http://software.intel.com/en-us/intel-compilers/.

4

Optimització

En aquesta secció explorarem les optimitzacions que podem aplicar al nostre codi per reduir el seu temps d'execució. Ordenades cronològicament, des del codi original fins a l'última optimització aplicada, en cada secció analitzarem el codi i discutirem una possible millora. Un cop implementada aquesta millora, estudiarem el nou rendiment del programa amb les eines presentades en la secció anterior. En cas que es redueixi el temps d'execució, adoptarem aquesta optimització al codi i avançarem a la següent.

Donarem les mesures obtingudes en el servidor i en el portàtil. Farem servir gfortran fins al moment en que necessitem el compilador d'Intel (i tornarem a mesurar el temps del codi original amb aquest compilador). Si no s'indica el contrari, les mesures de temps es fan amb GNU time, garantint un ús de 98% o més d'ocupació del processador. En els punts on la comanda time no sigui suficient, donarem mesures obtingues amb altres eines. Les mesures es repeteixen tres vegades i es fa la mitja dels resultats, amb excepcions del problema amb l'entrada més gran, ja que la gran quantitat de temps que necessita per executar-se limita l'obtenció de mesures.

A més de les gràfiques incloses en els diferents apartats, hem inclòs els temps mesurats en l'annex B.

4.1 Codi original I

En aquest apartat, analitzem el comportament del codi tal i com l'hem rebut del seu desenvolupador. Més endavant, en l'apartat 4.6, veurem que és neces-

sari modificar el codi original per generar nous resultats. Aquests nous resultats respecten la idea que hi ha darrera el programa. Finalment, en la secció 4.8, obtindrem les mesures fent servir el compilador d'Intel amb el codi produït a la secció 4.6.

Abans de començar a optimitzar, és important saber quan tarda el programa original en executar-se, per saber si realment és necessària una optimització del codi. En cas afirmatiu, fem també una anàlisi amb més profunditat del codi per detectar quines parts del programa ocupen més temps al processador. Com hem vist a l'apartat 1.4, quan parlàvem de la llei d'Amdahl, si apliquem les millores ens els punts crítics, el benefici que obtindrem en el rendiment global del programa serà major. Mirem primer el rendiment al servidor:

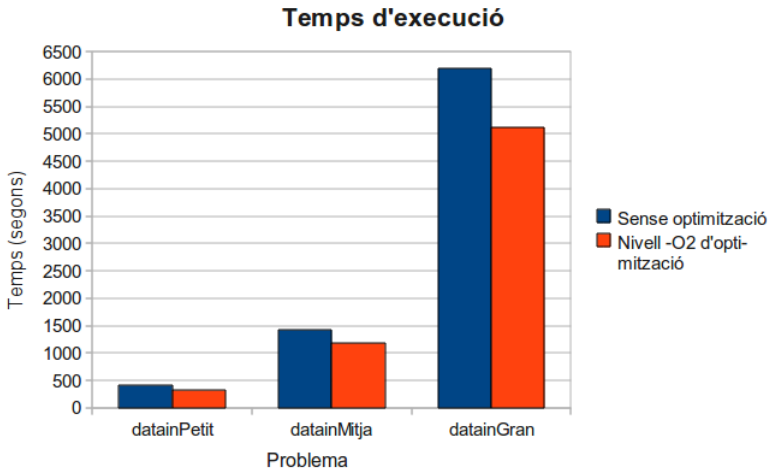


Figura 4.1: *Temps del programa original compilat amb gfortran i executat al servidor*

I el rendiment al portàtil:

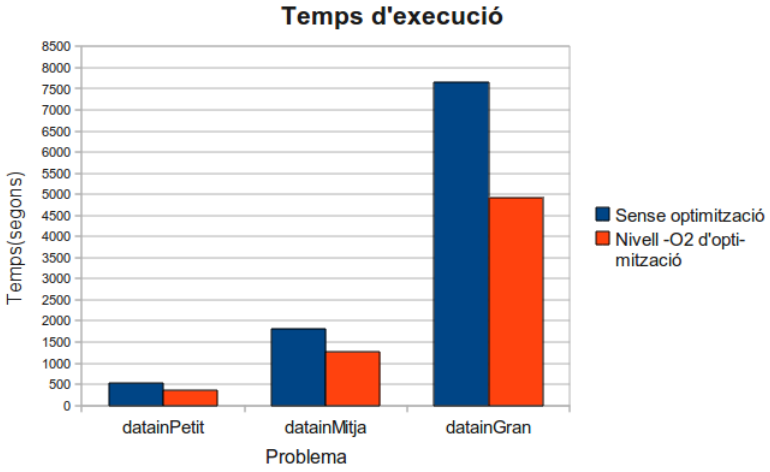


Figura 4.2: *Temps del programa original compilat amb gfortran i executat al portàtil*

Com podem veure, els temps són prou grans com per fer l'espera de resultats força llarga (fins a 2h, 7660 segons, al portàtil). El nostre objectiu és reduir aquest temps d'espera.

Procedim ara a fer una anàlisi de l'execució del programa amb eines més precises, per veure on el processador passa més temps executant operacions. Fem servir gprof amb el problema de mida mitjana i al portàtil. El resultat, a nivell de rutina, és:

```
gprof -b -p psv_ssg2_homog.pg
Flat profile:

Each sample counts as 0.01 seconds.
   %   cumulative   self              self     total
 time   seconds   seconds    calls   s/call   s/call  name
 27.02    182.37   182.37     2907     0.06     0.06  uxwz_
 25.34    353.41   171.03     2907     0.06     0.06  uzwx_
 14.45    450.94    97.53     2907     0.03     0.03  dxtxx_dztxz_
 12.49    535.22    84.28     2907     0.03     0.03  dxtxz_dztzz_
 12.16    617.26    82.05     2907     0.03     0.09  interior_
  8.43    674.13    56.86     2907     0.02     0.14  txxtzztxz_
  0.11    674.90     0.77     2907     0.00     0.00  freesurface_
  0.00    674.93     0.03        1     0.03   674.93  MAIN__
  0.00    674.93     0.00      969     0.00     0.00  outfs_
  0.00    674.93     0.00        1     0.00     0.00  div_coeff_
  0.00    674.93     0.00        1     0.00     0.00  grad_coeff_
  0.00    674.93     0.00        1     0.00     0.00  model_
```

Podem veure que gairebé el 78% del temps d'execució l'acumulen les rutines que calculen derivades (*uxwz_, uzmx_, dxtxx_dztxz* i *dxtxz_dztzz*). A més, un 12% del temps s'inverteix en la subrutina que genera les matrius U_{new} i W_{new} a partir de les derivades dels tensors, i un 8% en la subrutina que calcula els tensors a partir de les derivades de U i W. La resta del temps s'ocupa amb inicialitzacions de variables i en fer càlculs a la superfície lliure.

És evident que hem de concentrar els nostres esforços en les subrutines que calculen les derivades i els nous valors de τ_{xx}, τ_{xz}, τ_{zz}, $Unew$ i $Wnew$. Però, tenen alguna cosa en comú aquestes rutines?

Totes elles fan recorreguts per les diferents matrius, assignant valors a cada posició. Aquests valors s'obtenen amb operacions de suma, resta i multiplicacions de valors d'altres matrius. Són aquests recorreguts per les diferents matrius prou eficients?

4.2 Fusió de bucles i reordenació dels accessos

A més de les característiques tècniques de la màquina, és important conèixer també els aspectes funcionals del llenguatge de programació amb el que estem treballant. Fortran, a diferència d'altres llenguatges com C o Java, guarda els valors de les matrius per columnes. Això vol dir que, en l'espai reservat per a una matriu M a la memòria, el primer element que es guarda és el que correspon a la posició $M[0][0]$, i el segon és el que correspon a la posició $M[1][0]$. El tercer és $M[2][0]$, el quart $M[3][0]$ i així successivament.

A l'hora de recórrer una matriu, és important de cara a reduir el temps d'execució avançar a la posició immediatament adjacent en memòria a la posició que acabem d'actualitzar. Això és així per una propietat específica de la memòria cache: el principi de localitat espacial. Normalment, quan operem amb estructures que emmagatzemen diversos valors, accedim a un valor determinat, i sovint a valors adjacents a aquest. Un exemple d'aquesta situació és els vectors, que es recórren accedint a tots els seus valors. Quan el processador necessita accedir a un valor que està emmagatzemat a la memòria principal de la màquina, aquest valor es porta amb els seus valors adjacents, i es guarda a la memòria cache. D'aquesta forma, si el processador torna a necessitar aquest valor o qualsevol dels valors adjacents, aquests estaran disponibles a la memòria cache i podran ésser servits al processador de forma molt més ràpida.

El programa, tal i com està programat inicialment, no aprofita la localitat temporal. Si bé és cert que alguns dels bucles recorren les matrius per columnes (els que deriven la matriu en qüestió respecte a la direcció vertical), la meitat d'ells no ho fa així, provocant molts accessos a memòria principal i desaprofitant la cache.
La primera optimització que fem, llavors, consisteix en fer que els bucles que calculen les derivades de U i W recorrin aquestes matrius per columnes. A més, aprofitem que l'espai d'iteracions és pràcticament el mateix per fusionar els bucles que formen les derivades de U, U_x i U_z. Anàlogament, fusionem els bucles que formen les derivades de W, W_x i W_z.

Com que desfer i tornar a fer els bucles és una feina pesada, ja que hi ha molts casos particulars, de moment només aplicarem aquestes optimitzacions al bucle que genera les derivades de U i W. Més endavant, si això millora el temps del sistema, podrem replicar la tècnica als bucles que calculen les derivades dels tensors.

Observem ara el resultat de l'optimització amb gprof i el problema de mida mitjana, també al portàtil:

```
gprof -p -b psv_ssg2_homog.pg
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls   s/call   s/call  name
 49.71    317.51   317.51     2907     0.11     0.11  dudw_
 15.16    414.33    96.82     2907     0.03     0.03  dxtxx_dztxz_
 13.14    498.25    83.93     2907     0.03     0.03  dxtxz_dztzz_
 12.99    581.22    82.97     2907     0.03     0.09  interior_
  8.89    637.97    56.75     2907     0.02     0.13  txxtzztxz_
  0.12    638.74     0.77     2907     0.00     0.00  freesurface_
```

Per més comoditat, s'han descartat les rutines que menys contribueixen al temps total. Podem veure que el temps acumulat d'execució ha baixat. La rutina *dudw_*, que ha reemplaçat a *uxwz_* i *uzwx_* del codi original, ara triga 318 segons, mentre que el temps acumulat de les dues rutines abans era de 353 segons. No és un impacte molt dràstic, però augmentarà quan ho apliquem a les rutines que deriven els tensors. El temps total (descomptant el temps de sistema operatiu, que gprof no mostra) és de 638.75 segons, mentre que l'original era de 674.93 segons.

Mesurant el temps amb la comanda time al portàtil, obtenim una mitja de 1288 segons, que podem comparar amb l'anterior mesura en aquesta màquina: 1818 segons. L'impacte és molt positiu, així que incorporem aquesta millora al codi.

4.3 Desenrollat de bucles

La forma de les operacions que realitzem sobre les matrius fa que no aprofitem del tot la localitat espacial de la qual parlàvem a l'apartat anterior. Per calcular les derivades necessàries a partir dels elements de les matrius, fem servir el patró plantilla o *stencil*, que hem introduït quan explicàvem com resol el model el nostre programa.

Si estudiem com funciona aquest patró, veurem que el que fa es obtenir el valor d'una posició de la matriu a partir de valors en la mateixa posició i posicions adjacents en altres matrius. Per exemple:

```
do i = 1, nx

Ux(i,j) = div51x*U(i,j-1) + div52x*U(i,j)
         + div53x*U(i,j+1) + div54x*U(i,j+2)
Uz(i,j) = div51z*U(i-2,j) + div52z*U(i-1,j)
         + div53z*U(i,j) + div54z*U(i+1,j)

enddo
```

En el primer cas, sabent que Fortran guarda les matrius per columnes, és evident que s'aprofita bé la localitat espacial: al accedir als valors en diferents columnes en la iteració i, en la següent iteració del bucle ja disposarem dels valors adjacents a cadascuna de les columnes (p. ex: $U(i+1, j-1)$, $U(i+1, j)$, $U(i+1, j+1)$, etc.). En la segona operació no aprofitem que hem portat aquests valors en les diferents columnes, ja que només treballem amb la columna actual ($U(i, j)$).

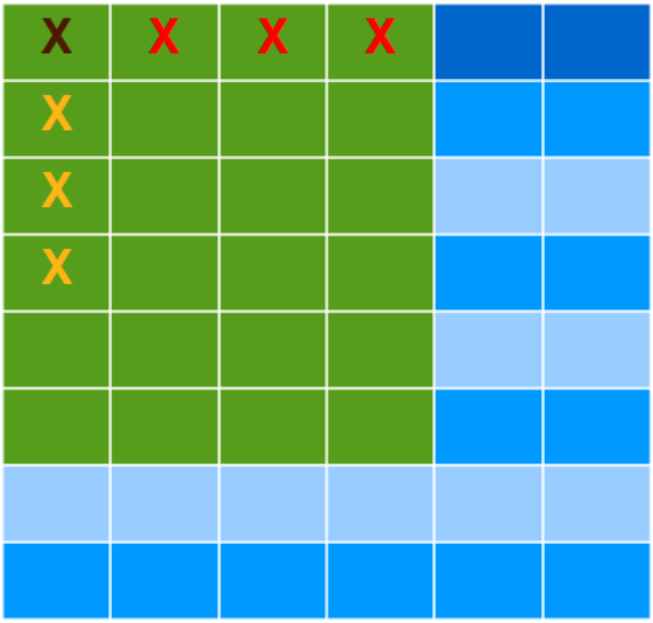


Figura 4.3: *Esquema que il·lustra el desaprofitament de localitat espacial*

En l'esquema representem una matriu amb l'*stencil* horitzontal en vermell i el vertical en groc. La zona de color verd representa les dades que hem portat a memòria cache al accedir als elements de les diferents columnes. L'element en negre representa la posició que tractem actualment.
Si apliquem un desenrollat de bucle, i fem que a cada iteració es facin les operacions per a j, $j+1$, $j+2$... aprofitarem millor les dades que tenim a la memòria cache.

Temps dels desenrollats				
j + 2	j + 4	j + 5	j + 6	j + 8
1286 s	1277 s	1280 s	1285 s	1293 s

Les mesures, tot i que algunes són positives, no són tan impactants com el cas anterior (recordem que el temps era de 1288 segons). De totes maneres, aquest desenrollat ens servirà després per vectoritzar les operacions, així que ho apliquem al codi amb desenrollat de 4, el qual ens ha donat el millor resultat.

4.4 Eliminació de matrius old

El fet que, segons la solució del model, necessitem els valors de les matrius U i U_{old} per obtenir els nous valors de U_{new} no ens obliga a mantenir una còpia de cada valor en una matriu separada. Si ens fixem en el codi, l'únic moment on necessitem els tres valors és en la rutina *Interior*, on es fa les següents operacions:

```
Unew(1:nzm1,1:nx) = 2.*U(1:nzm1,1:nx) - Uold(1:nzm1,1:nx)
Wnew(1:nz,1:nxm1) = 2.*W(1:nz,1:nxm1) - Wold(1:nz,1:nxm1)
```

I si, en comptes de guardar el valor antic a U_{old}, el guardem a U_{new}? Així ens estalviem dues matrius que poden ser molt grans (en el problema de mida mitjana són 1400x1400 posicions, cadascuna de les quals és un real de 4 *bytes*. En total, 7.8 MB cada matriu). Tot i que de moment aquesta millora no té un impacte significatiu en el temps d'execució, reduïm l'ús de memòria del programa i més endavant, quan modifiquem el codi per aprofitar millor la cache, serà positiu no fer servir tanta memòria.

Aquest canvi implica desenrollar el bucle principal per continuar oferint els mateixos valors que abans, i eliminar també la rutina $OutFS$. Amb la confirmació per part de l'autor del codi que aquesta rutina és il·lustrativa i no necessària pel resultat final, procedim a eliminar les matrius U_{old} i W_{old}.

Ara és un bon moment per avaluar els temps que hem aconseguit fins aquest moment. Els temps al servidor:

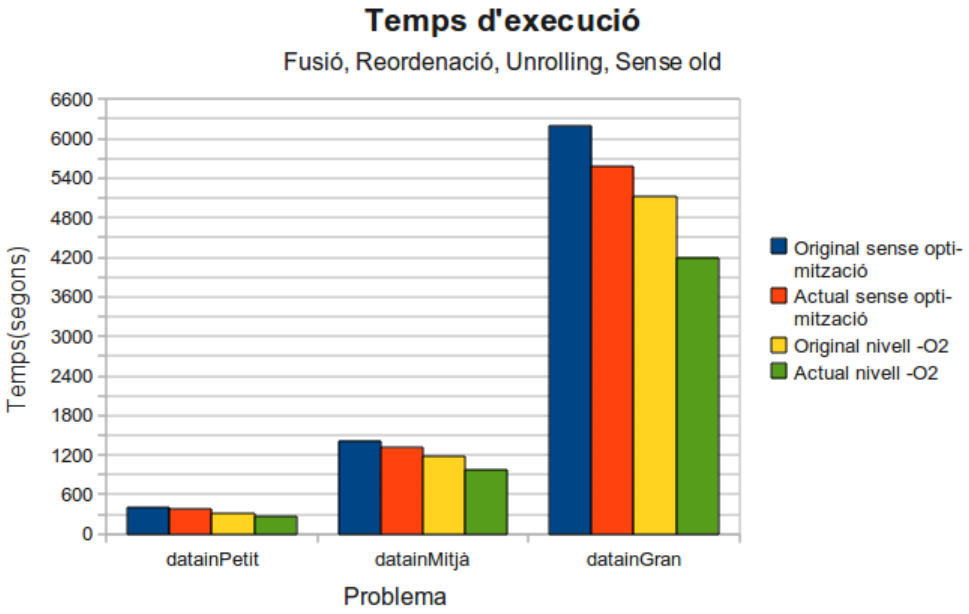


Figura 4.4: *Temps del programa actual compilat amb gfortran i executat al servidor*

i al portàtil:

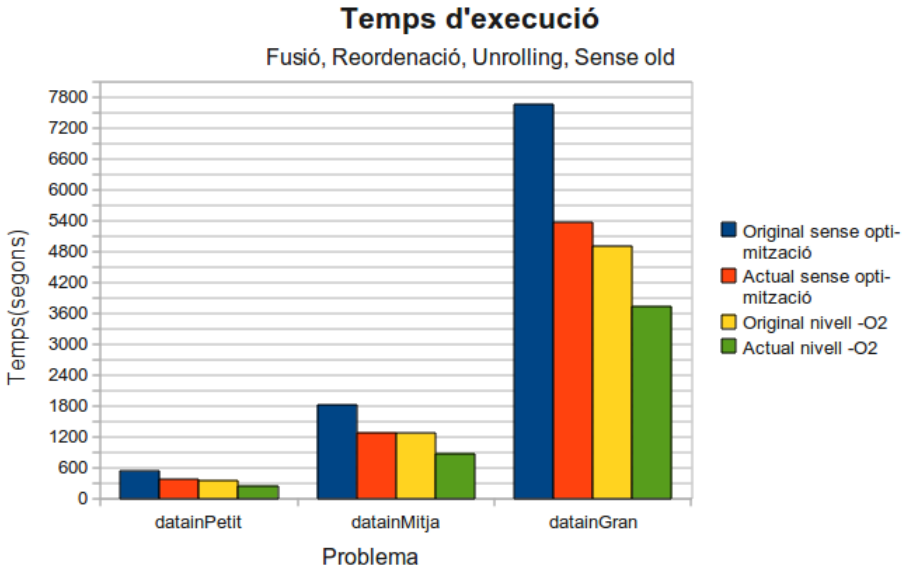


Figura 4.5: *Temps del programa actual compilat amb gfortran i executat al portàtil*

Un bon avanç, sobretot si tenim en compte que només ho hem aplicat a la meitat de les derivades (les de U i W). En el següent apartat aplicarem aquestes optimitzacions a la resta de derivades.

4.5 Extensió de les optimitzacions actuals a la resta d'operacions

En vista que els canvis que farem en el següents apartats a l'estructura de l'algorisme canvien radicalment la forma d'executar el codi, i sabent que haurem d'introduïr noves funcions i rutines en el programa principal, en aquesta versió decidim encapsular les funcions que calculen les derivades en mòduls independents de codi, que es troben en els fitxers 'DerUW.F90' i 'DerTxxTzzTxz.F90'. Això ho fem, a més, perquè el desenrollat que hem aplicat abans fa que el codi creixi molt i sigui difícil de corregir en cas d'error si no apliquem una mica de modularitat.

Aprofitant, a més, que ara les rutines que calculen derivades (que, recordem, són les més costoses en temps de càlcul) es troben en fitxers diferents, decidim aplicar les optimitzacions que tenim fins ara en la derivació de U i W (reordenació dels recorreguts de les matrius i desenrollat de bucle) a la derivació dels tensors.

Els temps que obtenim ara, amb el portàtil i el problema de mida mitjana són:

```
1089.49 user
116.42 system
20:05.91 elapsed
99%CPU
```

Són millors, però sembla que el temps de sistema ha crescut en comparació amb la versió anterior:

```
1191.24 user
89.82 system
21:21.12 elapsed
99%CPU
```

Aquest creixement mitiga l'impacte positiu de la nostra optimització.

Per determinar les causes de l'increment del temps de sistema, executem les dues versions del programa amb l'eina *ltrace*, que mesura el temps i la quantitat de crides a les rutines de les llibreries dinàmiques (les quals fan les crides a sistema). Els resultats (abreujats) de la versió anterior són:

```
ltrace -c ./psv_ssg2_homogGFOR
% time     seconds  usecs/call     calls      function
------ ----------- ----------- --------- --------------------
 86.12   49.873621        2910     17449 free
  7.74    4.285864         737      5814 expf
  4.68    2.558433         146     17449 malloc
  1.16    0.598545          25     23256 _gfortran_internal_pack
  0.11    0.064685          30      2127 _gfortran_transfer_real
  0.08    0.047746          15      3163 _gfortran_transfer_character
  0.04    0.021393          17      1228 _gfortran_st_write_done
  0.04    0.020580          16      1228 _gfortran_st_write
  0.03    0.018493          17      1047 _gfortran_transfer_integer
```

I de la versió actual:

```
ltrace -c ./psv_ssg2_homogGFOR
% time     seconds  usecs/call     calls      function
------ ----------- ----------- --------- --------------------
 94.42   57.881832        2488     23263 free
  4.45    2.729148         117     23263 malloc
  0.66    0.407469          17     23256 _gfortran_internal_pack
  0.21    0.125679          21      5814 expf
  0.08    0.050115          23      2127 _gfortran_transfer_real
  0.08    0.046670          14      3163 _gfortran_transfer_character
  0.03    0.020838          16      1228 _gfortran_st_write_done
  0.03    0.020090          16      1228 _gfortran_st_write
  0.03    0.017619          16      1047 _gfortran_transfer_integer
```

Com podem veure, en ambdues versions es fan moltes crides a la rutina *free*, que el que fa és alliberar memòria reservada. Això pot estar causat perquè fem servir variables locals de mida gran. Efectivament, si consultem les subrutines *TxxTzzTxz* i *Interior*, veurem que declarem matrius auxiliars que tenen la mida de les matrius senceres que fem servir. A més, en aplicar la fusió i reordenació dels accessos a les rutines que deriven els tensors, hem afegit dues matrius locals

més, la qual cosa explica aquest increment en temps del sistema.

Per sort, la solució és tan fàcil com declarar les matrius auxiliars com variables globals, i sobreescriure-les sempre que ho necessitem. Així ens estalviem tantes crides a les rutines *malloc* i *free*.

Finalment, els temps al servidor són:

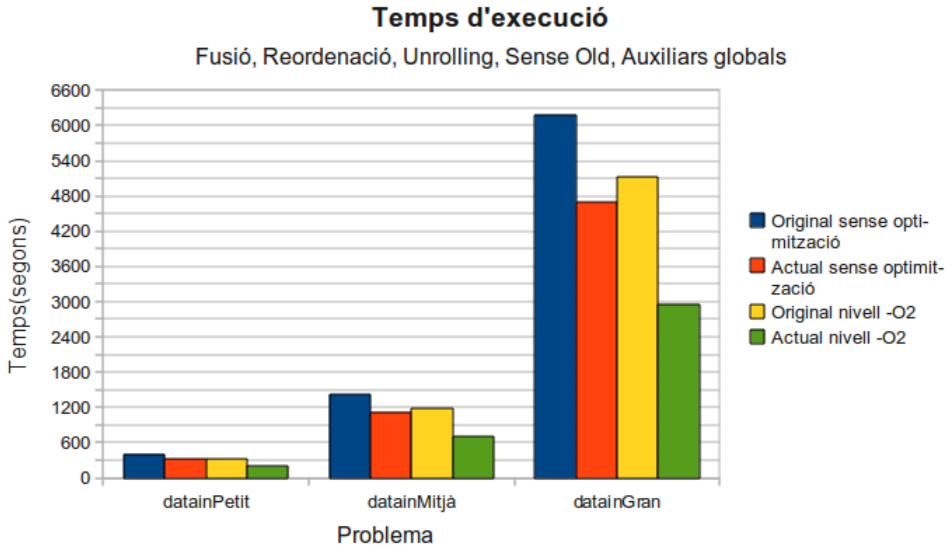


Figura 4.6: *Temps del programa actual compilat amb gfortran i executat al servidor*

i al portàtil:

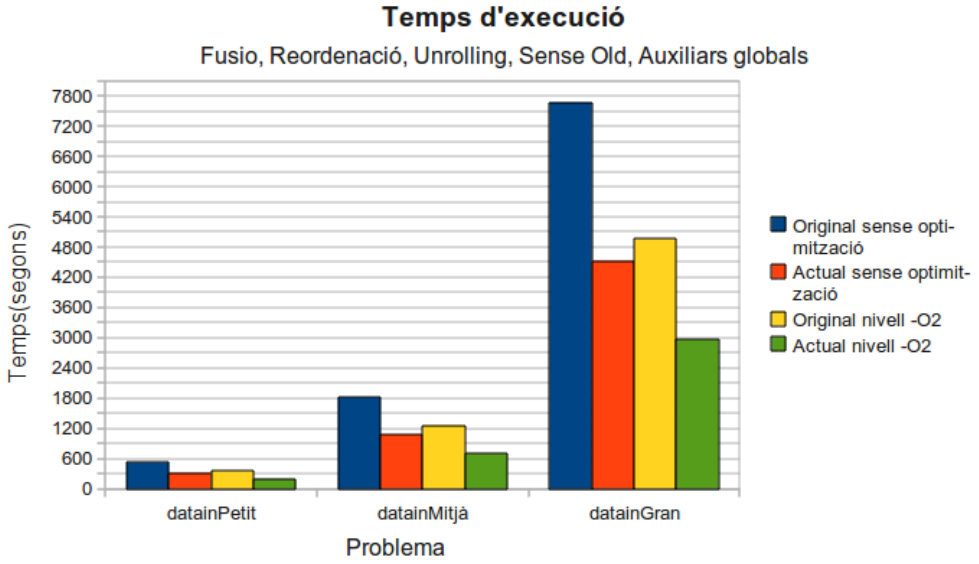


Figura 4.7: *Temps del programa actual compilat amb gfortran i executat al portàtil*

Com podem veure, el temps del nostre codi actual compilat sense l'opció '-O2' del compilador ja és menor que el temps del codi original optimitzat amb '-O2'.

4.6 Codi original II

Per evitar el desenrollat del bucle principal en el nostre codi, causat per l'eliminació de les matrius old, hem consultat a l'autor del mateix si és viable modificar els valors que el programa dona com resultat (el resultat són tots els valors de les matrius U i W, però, tal com està, el codi escriu només els valors d'una posició, ($izsou$,$jxsou$)) cada dos iteracions de la simulació en comptes de cada tres. L'autor ens ha confirmat que no hi ha problema amb fer-ho així.

Si modifiquem el programa perquè mostri els resultats cada dues iteracions no podem comparar ni els resultats ni les mesures de temps amb l'original que fèiem servir fins ara, perquè no fa exactament el mateix. És per això que re-dissenyem el programa original per tal que mostri els resultats cada dos iteracions. Al igual que fèiem abans amb el codi optimitzat, és necessari desenrollar el bucle principal del codi original per fer això. Aquesta versió del codi original és la definitiva i

amb la qual compararem tots els valors i mesures que fem d'ara en endavant.

Els resultats de les mesures del temps són pràcticament idèntics als de el codi original. Tot i així, els mostrem en l'annex B

4.7 Divisió en subproblemes

El programa, tal i com el tenim estructurat ara, no ens ofereix moltes oportunitats per explotar la memòria cache. Això és conseqüència de recórrer les matrius senceres, fent operacions, per produir noves matrius. Si les malles amb les quals treballem són grans, en el moment en que calculem els últims elements de la nova matriu, els primers que ja hem creat s'han fet fora de la memòria cache per deixar lloc als elements del final de la matriu.
Això és especialment cert en el servidor, on disposem d'un segon nivell de memòria cache, més petit que el tercer (256 KB en comptes de 8 MB), però molt més ràpid. El primer nivell és més petit, i en general ens serveix per les operacions que estem realitzant en un moment determinat (recorregut de bucles, etc.). En el tercer nivell pràcticament podem guardar una matriu sencera, però recordem que treballem amb diverses a la vegada. Per això, és important aprofitar aquests dos nivells (L2 i L3) de ràpida memòria cache.

A més, tal i com treballa ara el codi, no hi ha subproblemes que es puguin tractar de forma més o menys independent. Això dificulta molt la posterior paral·lelització. Si en un futur volem que el nostre codi es pugui dividir en parts que s'executin en paral·lel, necessitem definir aquestes parts.

Per tot això, en aquesta secció discutirem la divisió de la feina en subproblemes.

4.7.1 Forma de la divisió

Si el nostre objectiu és dividir el problema en parts més petites, que possiblement es puguin executar de manera independent entre elles, i estem treballant amb matrius, la primera idea que tenim és dividir la matriu en parts de similar mida, i operar cadascuna d'aquestes parts per separat. Aquest mètode s'anomena treballar amb rajoles o *tiles*.

Idealment, si podem dividir les nostres matrius en submatrius, independents les unes de les altres, podem treballar amb àrees de dades petites, realitzant les operacions una iteració darrera una altra fins acabar amb la rajola actual i passar a la següent. Això no només ens permet aprofitar millor la cache, sinó que a més, si disposem d'una màquina amb diversos nuclis, cada nucli pot treballar amb una *tile* diferent, explotant així el paral·lelisme del problema.

La realitat, però, no és tan senzilla. Les operacions del patró plantilla o *stencil* fan molt difícil la divisió de la matriu en parts independents. Com que necessitem els elements adjacents a l'element que volem calcular actualment, els elements que es troben als límits de les nostres rajoles necessiten elements que es troben a la rajola adjacent. És a dir, que les parts en les quals dividim el problema no són independents.

És, llavors, impossible dividir el nostre problema en subproblemes més petits? No necessàriament. El que podem fer es treballar amb *tiles* que calculen valors correctes i incorrectes (els que accedeixen a valors fora de l'àrea o "finestra d'operació"). Si modifiquem, entre operacions, la posició de la finestra, per incloure elements correctes ja calculats i deixar fora elements erronis, podrem treballar amb parts de forma relativament independent.

Si ens imaginem la matriu com una figura rectangular plana, i fem que la tercera dimensió, l'alçada, sigui el temps (que, en el cas del nostre programa, són les iteracions que van avançant), podem interpretar les idees anteriors com a figures geomètriques.
Amb una divisió perfecta, sense dependències entre rajoles, les figures serien prismes rectangulars; en canvi, si treballem amb una finestra, desplaçant-la per mantenir els valors correctes, les figures serien paral·lelepípede oblics.

Podem veure uns esquemes a la pàgina següent.

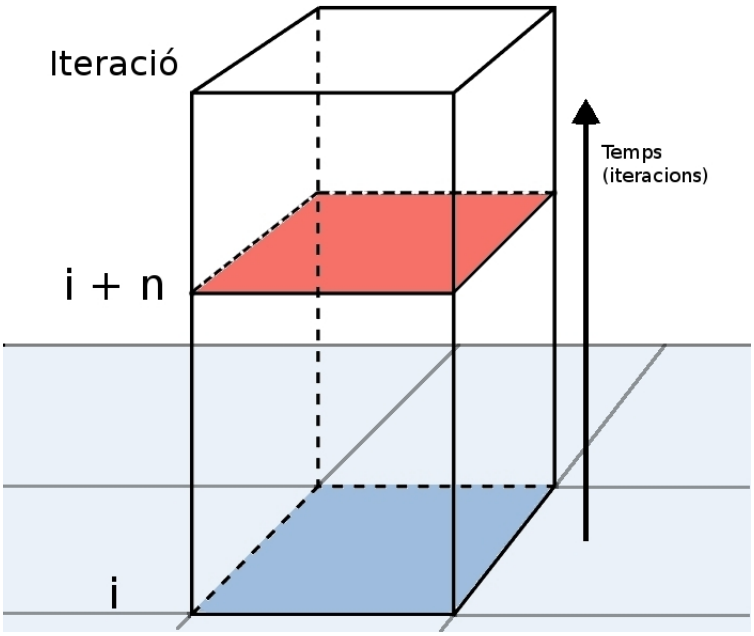


Figura 4.8: *Divisió sense dependències. La zona blava representa la secció de la matriu amb la iteració actual, la vermella la mateixa zona n iteracions després*

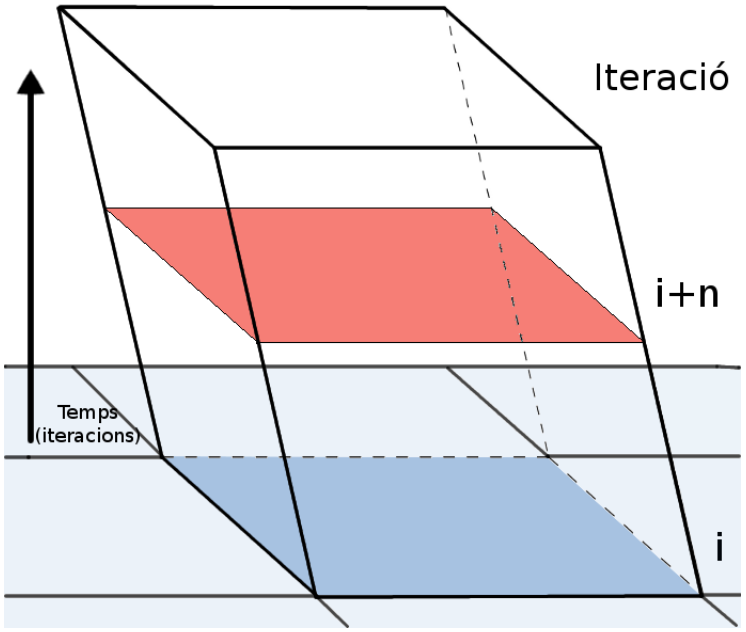


Figura 4.9: *Divisió amb dependències. La zona blava representa la secció de la matriu amb la iteració actual, la vermella la mateixa zona n iteracions després*

Imaginem que ens trobem en un punt intermedi de l'execució del programa. Hem de tractar la primera part en la que es divideix la matriu: la cantonada superior esquerra (per exemple). Si agafem una primera submatriu d'una mida z x x, i realitzem les operacions del codi sobre els elements, podem garantir que tots els elements, excepte els dels costats (per tractar-se de la cantonada superior esquerra, serien els costats dret i inferior) són correctes, ja que aquests accedeixen a elements fora de l'espai considerat amb les operacions de *stencil*. Descartem els valors incorrectes: això implica reduir l'àrea considerada, retirant la darrera columna i fila (o columnes i files, depenent de l'ordre de l'*stencil*).

Si ara avancem a la següent iteració i repetim les operacions sobre la nostra àrea reduïda, ens trobarem en la mateixa situació: els elements dels costats dret i inferior són incorrectes, així que els descartem. Repetim el procés durant n iteracions, i obtindrem una matriu més petita que l'original: $(z - s * n)$x$(x - s * n)$ on s indica els elements per la dreta i per sota de l'element actual que necessitem en l'*stencil*. Si veiéssim l'evolució com les figures geomètriques de la pàgina anterior, aquesta formaria una espècie de tronc de piràmide amb dos costats en angle recte amb la base.

En el cas de les rajoles interiors, aquestes no es fan més petites perquè no es troben en un dels costats de la matriu original. El que fan és desplaçar-se en dues direccions: si hem començat per la cantonada superior esquerra, les direccions són la vertical i la horitzontal, ambdues en sentit negatiu (és a dir, cap a la superfície lliure i el costat esquerra, o el que és el mateix, cap a l'element en la primera posició). El prisma rectangular original s'inclina: els costats esquerra i superior es "recolzen"sobre els elements ja calculats de la submatriu anterior. Així, accedim a valors correctes, que ja ha calculat la rajola anterior i no ha esborrat, perquè també es desplaçava en la mateixa direcció que l'actual. Mentrestant, els costats dret i inferior es deixen fora (amb valors correctes) quan la següent iteració hagués esborrat aquests valors correctes amb incorrectes. Així es forma el paral·lelepíped que hem explicat.

La inclinació és més o menys pronunciada depenent del nombre d'elements de l'operació de plantilla (*stencil*).

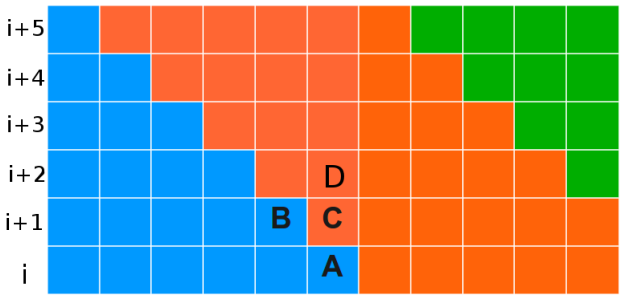


Figura 4.10: *Esquema il·lustratiu de les fronteres entre subproblemes*

Suposem un *stencil* on només necessitem un element adjacent per la dreta i per l'esquerra. En l'esquema, les files representen diferents iteracions. Quan estem treballant amb la submatriu blava, només podem garantir que els valors són correctes fins a la posició A. Per formar C, necessitem el valor adjacent per la dreta d'A, així que només podem formar fins a B.

Més endavant, quan estem treballant amb la submatriu taronja, per formar D necessitarem el valor B, i el tindrem, perquè gràcies a la inclinació del paral·lelepíped, aquesta posició no s'haurà esborrat, i el valor encara serà correcte.

Així, finalment, totes les divisions de la matriu original acaben encaixant i els valors que contenen són correctes.

La idea de treballar amb submatrius que, en el marc de les iteracions, es transformen en prismes està extreta de [8]. La diferència amb el model presentat en l'article és que el model de l'article és per treballar amb sistemes on la memòria no és compartida, (o és difícil de compartir), com pot ser una targeta gràfica o un sistema distribuït. Així, en aquest model, en comptes de fer servir paral·lelepípeds oblics, opta per repetir els càlculs de les zones on les submatrius limiten (on els resultats poden ser incorrectes), formant troncs de piràmide superposats.
Seria interessant, de cara a futurs treballs, implementar aquesta tècnica i mesurar el rendiment en sistemes distribuïts o amb targetes gràfiques.

4.7.2 Creació de la rutina que treballa amb submatrius

L'objectiu és, llavors, dividir les matrius en submatrius de menor mida, i treballar amb elles com hem explicat a l'apartat anterior. En aquest apartat, creem la

subrutina que, donada una submatriu, treballa amb ella executant n operacions abans de passar a la següent (subrutina `Tile()`).

Atès que les rajoles que es troben als límits laterals de les matrius es van fent més petites a mesura que s'executen les iteracions de la simulació, després d'un cert nombre d'aquestes iteracions és necessari detenir el procés i passar a la rajola següent. Un cop la resta de submatrius es trobin en el mateix estat que la que hem detingut abans, podem tornar a començar el procés. Això és així perquè si fem massa iteracions amb una submatriu lateral, al final no ens quedaran posicions per fer càlcul. I hem d'esperar a resoldre la resta de submatrius abans de tornar a agafar una altra vegada la base (que és més gran que els valors que hem calculat correctament) per evitar treballar amb valors incorrectes.

Tot i que en aquesta versió hem creat aquesta rutina, de moment encara li enviem la matriu sencera per que la executi, ja que no hem fet els canvis necessaris per incorporar el desplaçament de la finestra d'operacions. Com que no esperem cap millora en el codi (potser fins i tot va més lentament, per l'*overhead* que suposen les contínues crides a la rutina `Tile()`), no mesurem el rendiment.

4.7.3 Execució amb submatrius

Ara sí, ja creada la subrutina que treballa amb parts de la matriu, podem reordenar el codi perquè treballi executant parts de la matriu. La idea, que ja hem introduït a l'apartat anterior és, començant per la cantonada superior esquerra, avançar per la matriu en blocs, agafant un bloc i realitzant les operacions de n iteracions, i passar al bloc següent. Avancem verticalment per la matriu.

Amb aquests canvis, apareixen nous paràmetres per configurar el comportament del programa. Ara podem definir el nombre de iteracions seguides que volem que es facin per cada submatriu, així com la mida inicial d'aquestes. Aquests paràmetres estan relacionats.

Si analitzem detingudament el programa, veurem que les operacions de plantilla apareixen tres vegades en el transcurs d'una iteració: quan derivem U i W per obtenir els tensors, quan derivem els tensors per obtenir U_{new} i W_{new}, i finalment quan calculem la superfície lliure, ja que derivem $U_{new}[0][i]$ per formar $W_{new}[0][i]$. Com que, en cada cas, l'*stencil* és de quart ordre, fem servir, com a màxim, dos elements adjacents de qualsevol direcció per obtenir l'element actual.

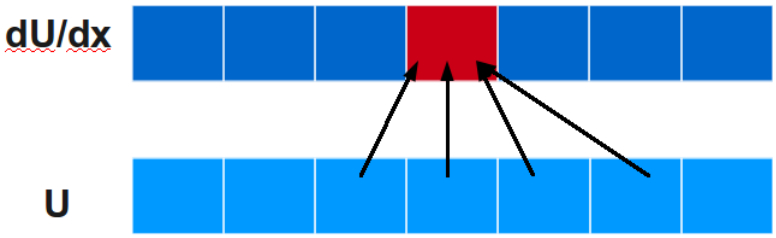


Figura 4.11: *Exemple de operació plantilla de quart ordre, amb dos elements per la dreta*

De tot això s'extreu que per cada iteració perdem 6 elements si ens trobem en una submatriu lateral, i ens desplacem 6 posicions cap amunt i cap a l'esquerra si estem en una submatriu interior.

Com que amb cada iteració perdem 6 elements en les submatrius laterals, i els elements de les matrius que estan situats als límits necessiten com a mínim 6 elements adjacents per formar-se, la mida mínima de la rajola al final d'una passada o grup d'n iteracions ha de ser de 6x6. Si perd 6 elements per iteració, podem calcular la mida de la submatriu o rajola inicial com:

$$xsize \geq n * 6 + 6$$

$$zsize \geq n * 6 + 6$$

(Nota: al codi, el nombre d'iteracions s'especifica amb la variable Num2Iter, però cada unitat representa dos iteracions. Així, $Num2Iter = 2$ són en realitat 4 iteracions)

Tal i com està ara, el codi ja funciona correctament. Tot i així, encara es pot millorar la gestió de l'*stencil*, especialment pel que fa a la superfície lliure i a la quantitat d'espai en memòria que ocupem a cada submatriu.

4.7.4 Eliminació de matrius auxiliars i diferents mides de submatrius

Per tal d'aprofitar al màxim la memòria cache que tenim, ens interessa reduir al màxim la quantitat de matrius auxiliars que fem servir. Si fem un recompte, veurem que fem servir 11 matrius en els càlculs: U, U_{new}, W, W_{new}, τ_{xx}, τ_{xz}, τ_{zz}, i les quatre matrius auxiliars per guardar les derivades direccionals. Això són

11 matrius de mida $zsize$ x $xsize$ repartint-se la memòria cache.
Si ens fixem en el segon nivell de memòria cache del servidor, aquest té 256 KB disponibles. Suposem que les submatrius són quadrades. Suposem també, que volem fer 12 iteracions seguides amb cada submatriu abans de passar a la següent; no és una quantitat molt gran, tenint en compte que en el problema de mida mitjana fem unes 2900 iteracions totals (1454 dobles cicles). A partir de les fórmules de la secció anterior, concloem que la mida mínim de la submatriu quadrada ha de ser de $12\text{x}6 + 6 = 78$ elements de costat, un total de 78^2 reals, cadascun dels quals ocupa 4 *bytes* a les nostres màquines. Així:

$$\frac{256 * 1024}{78^2 * 4} = 10.77$$

No tenim prou espai a la cache per totes les matrius. És cert que en les matrius laterals el tamany es va fent més petit. Però també voldrem fer més de 12 iteracions seguides amb cada matriu per aprofitar més les dades en cache i reduir el cost que suposa canviar de submatriu amb molta freqüència.

Quines matrius són imprescindibles? Sabent com es fa l'*stencil*, la solució està clara: les matrius U, U_{new}, W, W_{new}, τ_{xx}, τ_{xz}, τ_{zz} guarden valors als quals accedim quan treballem amb *tiles* adjacents: no té sentit eliminar-les si després hem de crear *buffers* matricials per guardar els valors. I les matrius auxiliars que guarden les derivades? En realitat, no són necessàries fora de la submatriu actual. Si modifiquem el codi, podem fer que desapareguin i els càlculs es facin directament sobre les altres matrius.

A més, tal i com està ara el codi, apliquem una reducció de 6 elements en els costats dret i inferior, a causa de les operacions de *stencil*. Perdem 2 elements per operació de derivació. Però només en la superfície lliure es fa una tercera derivada, quan es deriva $U_{new}[0][i]$ per formar W_{new}. És, llavors, imprescindible aplicar una reducció de 6 elements per iteració a totes les submatrius, quan només derivem tres vegades en les submatrius superiors i dues a la resta?

Una reducció en el nombre d'elements que es perden per iteració en cada submatriu ens permetria fer més iteracions amb la mateixa mida de la submatriu inicial. El càlcul del tamany mínim de les submatrius que no inclouen la superfície lliure seria:

$$xsize \geq n * 4 + 6$$
$$zsize \geq n * 4 + 6$$

Conseqüentment, amb els mateixos elements podem fer més iteracions. Les rajoles superiors hauran de ser més grans, per fer el mateix nombre d'iteracions,

però només en el sentit de les x, que és la direcció de la derivada que es fa per calcular la superfície lliure.

Fent el mateix càlcul que abans, les mateixes submatrius de 78x78 elements serien suficients per 18 iteracions, i si tenim 256 KB de memòria cache:

$$\frac{256 * 1024}{78^2 * 4} = 10.77$$

com abans, però com que ara només tenim 7 submatrius, cabrien totes senceres a la memòria cache i quedaria lloc per altres variables.

4.7.5 Ajust dels paràmetres

Un cop re-dissenyat el programa perquè treballi amb divisió en subproblemes, només ens queda ajustar els nous paràmetres i fer les proves pertinents per seleccionar la millor versió.

Les proves les realitzem al servidor, que és on podem treure profit del nivell mitjà de cache. Executem el programa amb el problema de mida mitjana, limitant el nombre d'iteracions a 1320 i modificant únicament la mida de les submatrius i el nombre de iteracions seguides que es fa amb una submatriu. Les mesures es repeteixen 3 vegades.

Ajust dels paràmetres		
Nombre d'iteracions	Mida	Temps
4	22x22	424 s
8	38x38	424 s
12	54x54	421 s
16	70x70	419 s
24	102x102	418 s
32	134x134	423 s
240	486x486	438 s

La variació entre nombres d'iteracions menor de 32 és similar, i només quan les submatrius creixen molt es disparen els temps. Això ens mostra que la cache

no s'aprofita del tot. El motiu principal és, probablement, que la finestra d'operacions, o submatriu actual, es va desplaçant per la matriu sencera a causa de les operacions d'*stencil*. Això, sumat a la poca reusabilitat de les dades (ja que, una vegada assignats els valors a una submatriu, aquesta només es llegeix una vegada abans de ser sobreescrita) fa que el programa no tregui massa rendiment de treballar amb subproblemes.

Així ho confirmen els temps.

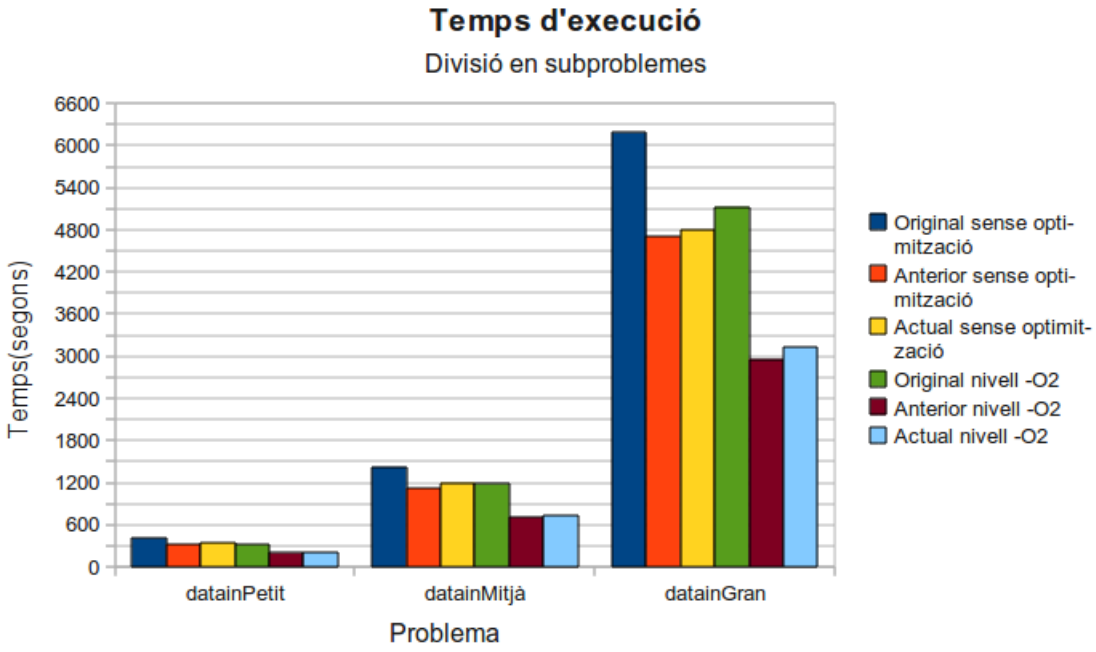


Figura 4.12: *Comparació de versions compilades amb gfortran i executades al servidor*

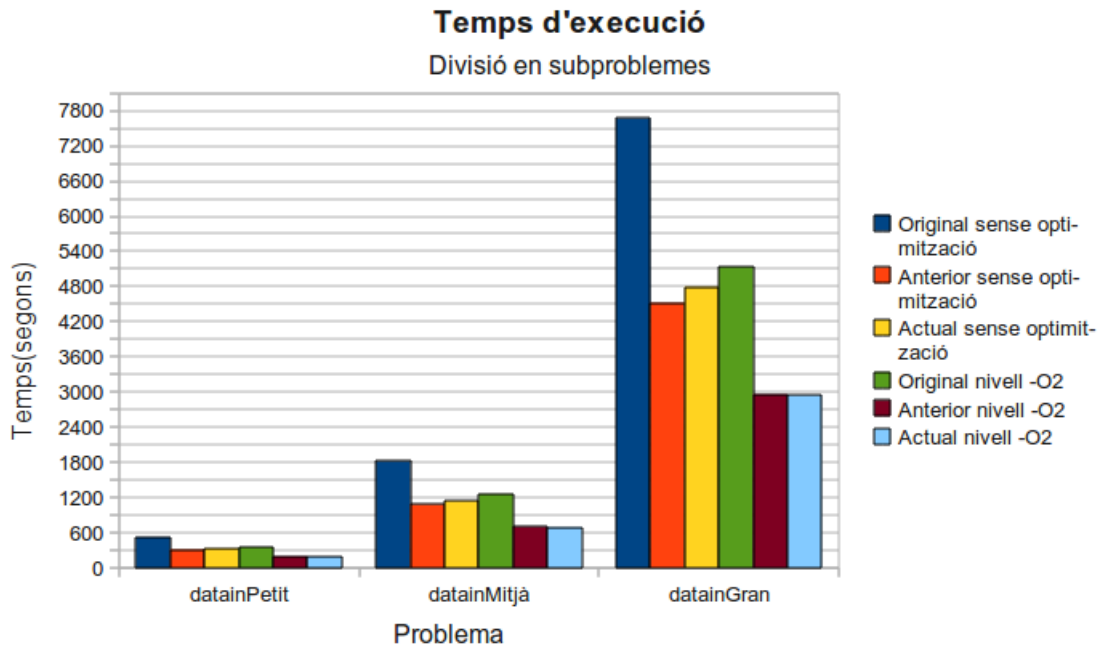


Figura 4.13: *Comparació de versions compilades amb gfortran i executades al portàtil*

En les gràfiques, 'Anterior' es refereix a la millor versió abans de dividir en subproblemes (que correspon a la presentada en l'apartat 4.5).

Els temps són una mica pitjors en totes les versions, excepte en el portàtil compilant amb l'opció d'optimització '-O2', on els temps són lleugerament inferiors. Ara cal valorar si val la pena aquesta pèrdua d'uns segons i continuar amb aquesta versió, o desfer els canvis i buscar un altre camí.

En general, els temps són pitjors, però no molt pitjors. En la majoria de casos són uns pocs segons de diferència, i en un d'ells fins i tot és més ràpid. I si considerem els beneficis que ens pot aportar el dividir el problema en parts (principalment, poder executar diferents parts en paral·lel), pensem que el benefici val la pena el risc i ens decidim a continuar millorant aquesta versió.

4.8 Codi original III

En el proper capítol explicarem el procés de vectorització de les operacions de la nostra aplicació. Veurem que per vectoritzar, ens són molt útils els compiladors d'Intel, ja que fan molta de la feina de manera automàtica.

A més de vectoritzar, el compilador d'Intel aplica una sèrie d'optimitzacions de forma automàtica. Per poder comparar la nostra versió amb l'original i mesurar el guany real, hem de compilar el programa original amb el compilador d'Intel per a Fortran, ifort, i mesurar els temps que ens dona.

Executem dues versions: la versió compilada només demanant precisió en el càlcul amb els reals (ja que, si no ho fem, els valors poden ser diferents dels que ens dona la nostra versió depenent de l'ordre exacte en el que s'executen les operacions), i la versió que no demana precisió. En aquest últim cas, serà necessari fer una comparació entre els resultats obtinguts i els que dona el model analític per poder valorar si la solució és prou precisa. Això es deixarà com a feina futura.

Recordem que, per defecte, el compilador d'Intel optimitza al nivell '-O2' de gfortran. Els resultats per al servidor són:

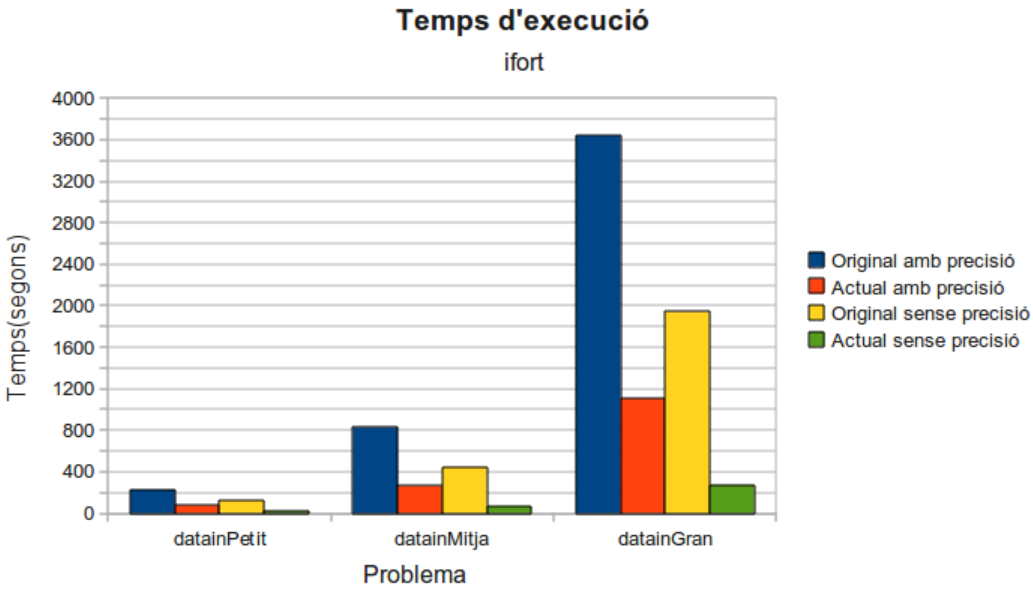


Figura 4.14: *Comparació de versions compilades amb ifort i executades al servidor*

I al portàtil:

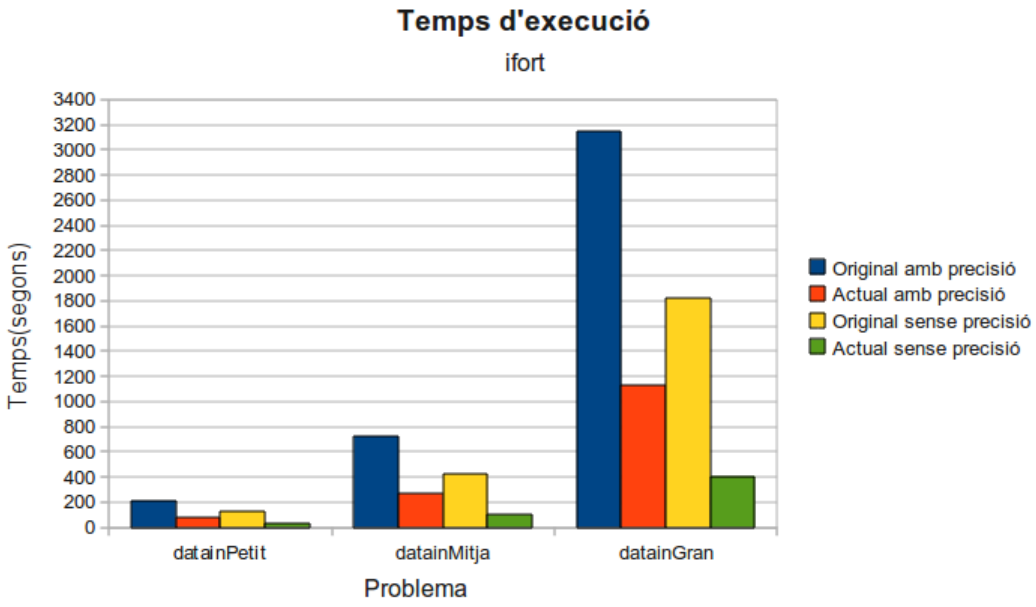


Figura 4.15: *Comparació de versions compilades amb ifort i executades al portàtil*

4.9 Vectorització

La vectorització consisteix en explotar un recurs que ofereixen els processadors per realitzar diverses operacions de forma simultània. Aquestes operacions s'anomenen operacions vectorials, i a diferència de les operacions escalars que normalment es fan servir, aquestes operen amb més de dos operands per produir múltiples resultats. Tot i que es pot considerar una forma de paral·lelisme, nosaltres ho incluim en l'apartat d'optimització, per distingir-lo d'altres formes de paral·lelisme de més alt nivell.

Les instruccions vectorials van aparèixer primer en supercomputadors com els CM-1 i CM-2 del Massachusetts Institute of Technology. Intel va popularitzar aquestes instruccions (també anomenades SIMD, o *Singe Instruction, Multiple Data*) al incloure-les en els seus processadors domèstics, en el conjunt MMX i posteriorment la sèrie SSE (*Streaming SIMD Extension*).

Quina és la idea que hi ha darrera de l'ús d'instruccions vectorials? Imaginem que tenim dos vectors, de quatre posicions cadascun, i volem obtenir la suma dels elements que corresponen a la mateixa posició en cada vector, guardant-la en un altre vector. Amb instruccions escalars, el que hauriem de fer és recórrer els vectors, sumant els valors de cada posició. Aquestes operacions es transformen en una simple suma amb les instruccions vectorials. I no només es poden fer sumes: hi ha desenes d'operacions disponibles per treballar amb registres vectorials (registres especials de 128 bits, que poden guardar múltiples valors, ja siguin caràcters, enters, reals, o reals de doble precisió).

Figura 4.16: *Esquema del funcionament d'una instrucció vectorial*

Aquestes instruccions són especialment útils quan el codi està format per gran quantitat de bucles que realitzen molts càlculs sobre vectors o matrius, com és el nostre cas. Malgrat tot això, també tenen algunes limitacions: sovint s'han d'introduir manualment en el codi, perquè no tots els compiladors les apliquen automàticament. A més, les dades han d'estar alineades a múltiples de 16 bytes per poder treure veritable profit a la vectorització.

4.9.1 Vectorització automàtica. Gfortran

Aqui ens trobem amb un problema fent servir el compilador de GNU per fortran, gfortran. El manual del compilador no diu res sobre la vectorització automàtica. Tot i així, en un apartat, el manual diu que gfortran accepta totes les opcions que accepta gcc, el sistema de compiladors de GNU del qual gfortran forma part.

Conseqüentment, decidim compilar el codi amb l'opció '-ftree-vectorize' de gcc, que activa la vectorització automàtica, i '-ftree-vectorizer-verbose=n', que hauria de mostrar els bucles que ha optimitzat el compilador. La compilació no mostra informació sobre la vectorització. Un cop compilat, analitzem el codi desensamblant l'executable amb l'eina **objdump**, que ens mostra el codi ensamblador del mateix. Apareixen algunes instruccions vectorials, sí, però són operacions escalars que treballen amb registres vectorials, com per exemple, 'addss', que suma un real de 4 bytes de la part baixa d'un registre vectorial a la part baixa d'un altre registre vectorial. No sembla que s'estigui aprofitant per fer operacions.

Per assegurar-nos, compilem el mateix programa, de forma idèntica pero sense l'opció de vectorització automàtica, i el desensamblem amb objdump. Comparant els dos resultats, podem veure que és exactament el mateix executable. L'opció de vectoritzat automàtic no ha afegit cap canvi. I si tenim en compte que al manual de gcc s'afirma que la vectorització automàtica s'activa per defecte si compilem únicament amb l'opció '-O3', podem deduir que no està aplicant vectorització, com a mínim no alterant el codi produit per millorar el temps.

4.9.2 Vectorització automàtica. ifort

El manual del compilador d'Intel sí fa referència a la vectorització automàtica. De fet, el compilador l'activa per defecte amb l'opció '-O2' (que, alhora, és l'opció per defecte). Si compilem amb aquest compilador, ifort, amb l'opció '-vec-report[n]', ens mostra quins bucles vectoritza de forma automàtica i quins no, especificant el motiu. Hem de tenir en compte, llavors, que els temps mostrats a l'apartat 4.8 inclouen la vectorització automàtica.

Però, és aquesta optimització la millor que es pot fer? Sembla que no, pels informes que ens mostra el compilador. Entre la miríada de missatges que ens mostra, trobem un que ens crida l'atenció:

```
ifort -xSSE4.1 -vec-report2 AuxMods.F90 DerUW.F90 DerTxxTzzTxz.F90
psv_ssg2_homog.F90 -o psv_ssg2_homogVECTOR
[...]
An internal threshold was exceeded for routine
dertxxtzztxz_mp_composeuwnew_ and optimization level may be
reduced. See HTTP://SOFTWARE.INTEL.COM/EN-US/ARTICLES/
INTERNAL-THRESHOLD-WAS-EXCEEDED for more information and advice.
[...]
```

Inmediatament després d'aquest missatge, s'atura la vectorització del fitxer DerTxxTzzTxz.F90. Si consultem la web indicada, ens trobem amb una solució, en forma d'opció del compilador, que només fa que la compilació del codi no acabi mai. En comptes d'això, mirem el programa a veure si podem detectar el problema.

No és d'estranyar que el compilador tingui problemes per optimitzar el codi del fitxer DerTxxTzzTxz.F90: són 1800 línies de codi, amb un gran nombre de bucles i bucles aniuats. Provem separant aquesta gran rutina en parts més petites, i tornant a compilar.

Ara el missatge desapareix. Sembla que el compilador ho fa prou bé: vectoritza gairebé tots els bucles vectoritzables. Té alguns problemes amb expressions massa complexes, però la resta, segons l'informe del compilador, està vectoritzat.

Tot i així, decidim fer una prova, canviant les variables escalars o les variables auxiliars on guardem resultats intermedis que es fan servir per calcular les derivades dels tensors, per vectors amb els mateixos valors que aquestes variables. Potser així el compilador pot detectar punts on fer servir les instruccions vectorials amb més facilitat.

El resultat és negatiu: el compilador detecta moltes dependències de dades, que ara s'emmagatzemen en vectors en comptes de en variables locals, i falla més a l'hora de vectoritzar. Així ho demostra els resultats d'executar la versió amb variables escalars i la versió amb variables vectorials:

```
Escalars:
ifort -xSSE4.1 -fp-model precise [...]
5:02.46 elapsed 99%CPU

Vectorials:
ifort -xSSE4.1 -fp-model precise [...]
5:13.58 elapsed 99%CPU
```

Resumint, no cal provar el mateix amb la rutina que deriva U i W, ja que només empitjorarem el temps. Ens quedarem amb la versió que treballa amb variables escalars, i separant el codi que calcula les derivades en parts per facilitar la feina d'autovectorització al compilador.

Podem trobar un resum de l'informe del autovectoritzat a l'annex C

4.9.3 Vectorització manual

Com ja hem introduït abans, les instruccions vectorials treballen millor si les nostres dades estan alineades a una direcció múltiple de 16 bytes. De fet, si intentem carregar o guardar una dada amb una instrucció vectorial a una posició de memòria no alineada a 16 bytes, el programa s'avortarà amb un error de segmentació.

Per solucionar aquest problema, es van crear en el seu moment unes instruccions per carregar i guardar dades a posicions de memòria no alineades. Aquestes instruccions, tot i facilitar-nos les coses, tenen un gran problema: són molt més costoses en temps que les instruccions equivalents amb direccions alineades.

Com que les direccions amb les que treballem amb les nostres submatrius depenen del moment de l'execució on ens trobem, el compilador d'Intel és incapaç de decidir, en el moment de compilar, si els nostres accessos seran alineats o no alineats. Per evitar provocar errors de segmentació, el compilador opta per l'opció més segura, que és la de fer servir instruccions per direccions no alineades.

Però, si ho pensem, les nostres submatrius o finestres d'operació es desplacen 4 elements verticalment entre iteracions. Aquest 4 elements són reals, cadascun dels quals ocupa 4 bytes, és a dir, ens desplacem 16 bytes. Si la direcció on comencem a treballar està alineada a 16 bytes, podrem treballar amb accessos alineats sense introduir ajustaments entre iteracions.

El nombre d'elements que ens desplacem horitzontalment no ens importa, sempre i quan el nombre de files total de la matriu sigui múltiple de 4 reals (16 bytes). Si la nostra posició (fila) està alineada en una columna, ho estarà en totes. Recordem que Fortran guarda els valors de les matrius per columnes: si les dades estiguessin declarades en un codi en C, seria el mateix cas però aplicat a files.

Per això els nostres experiments tenen unes malles amb un nombre de files igual a $4n-1$ (treballem de [0..4n-1], en total un nombre d'elements múltiple de 4). No creiem que sigui una imposició massa estricta: sempre es poden trobar múltiples de 4 propers al valor de la mida de la malla que volem.

Tot això ens permet vectoritzar a ma els càlculs de cada submatriu, canviant els accessos a memòria no alineada per accessos alineats.

Per fer això, farem servir unes funcions intrínseques programades en C que faciliten l'ús d'instruccions vectorials, oferint-nos instruccions d'alt nivell que encapsulen les instruccions en codi màquina. Aquest canvi ens obligarà, llavors, a barrejar codi C amb Fortran. El codi C serà compilat amb el compilador d'Intel per C/C++, **icc**.

Malgrat les direccions inicials estaran ara alineades, les operacions de plantilla ens forcen a accedir a posicions no alineades amb càlculs com:

```
DzTxz = g51z*Txz(i-1,j) + g52z*Txz(i,j) + g53z*Txz(i+1,j) + g54z*Txz(i+2,j)
```

Per resoldre aquests casos, és necessari accedir a posicions no alineades, amb les instruccions més lentes. No obstant, hem notat una lleugera reducció de temps si fem servir la instrucció shuffle per carregar un dels valors. L'instrucció shuffle col·loca dos valors d'una variable vectorial i dos d'una altra variable vectorial en una tercera variable vectorial. Així, si hem carregat `Txz(i,j)` i `Txz(i+2,j)`, podem obtenir `Txz(i+1,j)` com:

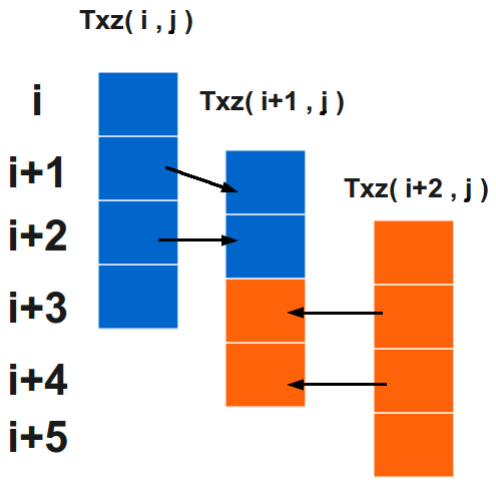


Figura 4.17: *Funcionament de la instrucció shuffle per carregar valors*

A continuació mostrem unes gràfiques amb la comparació entre la millor versió autovectoritzada i la versió vectoritzada a mà:

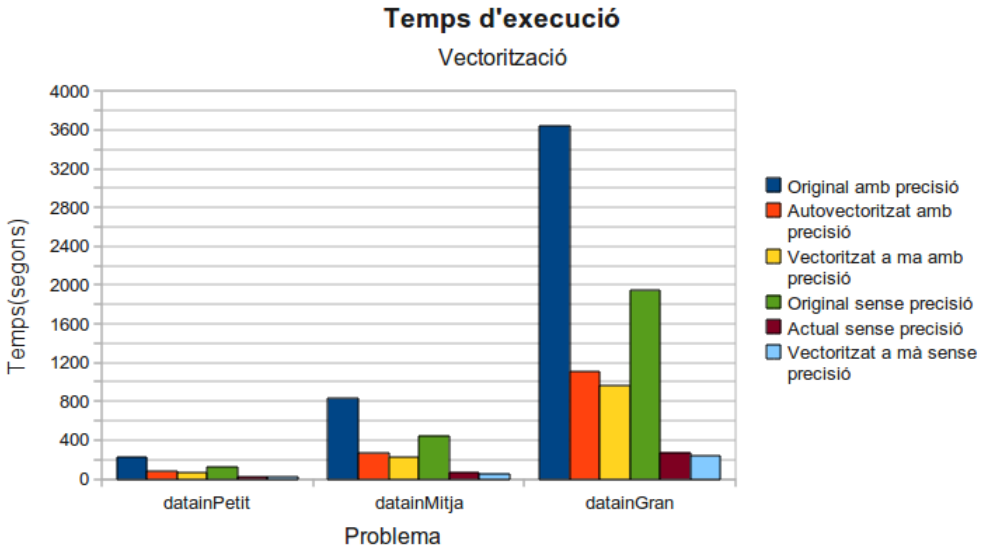


Figura 4.18: *Comparació de versions vectoritzades i executades al servidor*

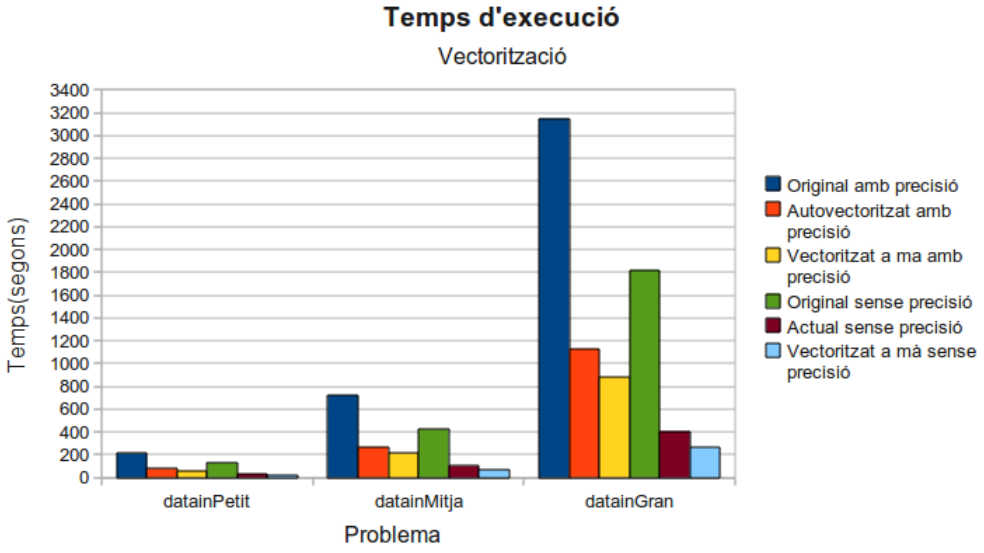


Figura 4.19: *Comparació de versions vectoritzades i executades al portàtil*

5

Paral·lelització

Un cop optimitzat el codi, ens centrarem en un altre aspecte clau per reduir el temps d'execució de la nostra aplicació: el paral·lelisme.

Avui en dia, molts dels ordinadors que es produeixen incorporen diversos processadors, o processadors amb diversos nuclis, o una combinació d'aquestes configuracions. Aquests ordinadors, al disposar de més d'una unitat de processat per realitzar càlculs, tenen més facilitats per executar múltiples tasques alhora (abans els processos s'executaven en sèrie, repartint-se el processador durant fragments petits de temps, de forma que es simula l'execució simultània; aquesta manera d'operar s'anomena concurrència en el processador). Això augmenta la fluïdesa i el rendiment de les nostres màquines quan estem fent servir diverses aplicacions a la vegada.

Però no només serveixen per executar millor múltiples aplicacions. Modificant la nostra forma de programar, podem millorar les nostres aplicacions, fins ara seqüencials, per a que treguin profit de la redundància d'unitats de càlcul: és a dir, paral·lelitzar els nostres programes. Quan paral·lelitzem, el que fem és detectar quines parts del programa es poden executar alhora, i enviem cadascuna d'aquestes parts a un nucli diferent perquè les calculi a la vegada, reduint així a la meitat, o a una tercera, quarta, cinquena... (depenent del nombre de nuclis) el temps d'execució.

A més, el tema dels ordinadors amb múltiples processadors respon també a un objectiu comercial. Els usuaris que fan servir regularment ordinadors, a l'hora d'adquirir un de nou, es guien per la potència i les prestacions aparents de la màquina: si les companyies venen ordinadors amb dos o més processadors, que "doblen la potència de càlcul", els consumidors es decantaran per una màquina d'aquestes característiques, encara que a la pràctica no treguin veritable profit

d'aquesta capacitat de rendiment addicional.

Sigui com sigui, els computadors amb diversos nuclis estan aquí per quedar-se. I aquesta nova arquitectura imposa nous paradigmes de programació per poder treure'n veritable profit. En general, ens hem d'acostumar a programar en paral·lel per millorar el comportament dels nostres programes en aquestes màquines.

Tota la paral·lelització que discutim aquí és per sistemes amb memòria compartida: això és, els nuclis es troben en una mateixa màquina o en un sistema en el qual accedeixen a la mateixa memòria. Totes les dades del programa són accessibles pels diferents fils d'execució. Per contra, en els sistemes distribuïts, l'execució de threads es delega a màquines independents o sistemes on no es comparteix la memòria. Ambdós sistemes tenen avantatges i desavantatges. Nosaltres tractem els sistemes amb memòria compartida perquè són els més habituals, i són el cas concret de les nostres màquines. No obstant, queda com a possible continuació de la feina el desenvolupar un paral·lelisme amb sistemes distribuïts.

5.1 Paral·lelització amb POSIX Threads

POSIX Threads és una interfície de programació per crear aplicacions que treballen amb múltiples fils d'execució o *threads*. Els threads es poden executar simultàniament en diversos processadors, oferint així el paral·lelisme que busquem.

POSIX Threads neix com un intent d'estandarditzar el concepte de fil d'execució, que cada venedor de hardware implementava a la seva manera, dificultant la portabilitat del codi entre màquines. És una de les interfícies de programació per paral·lelisme més antigues. La versió de sistemes UNIX està especificada amb l'IEEE POSIX 1003.1c estàndard de 1995. També és una de les més potents, ja que permet dissenyar amb precisió l'aplicació i el comportament dels fils d'execució.

Com a aspecte negatiu s'ha de dir que, a conseqüència d'aquest potencial per dissenyar l'aplicació com nosaltres volem, també és una de les que més esforç requereix per implementar aplicacions amb una certa mida, ja que moltes de les funcionalitats (cues de fils, sincronització, etc.) les hem de dissenyar nosaltres a mà. Aquest serà un dels motius, com veurem després, que ens farà decantar-nos

per altres interfícies.

5.1.1 Implementació

Per implementar el nostre programa amb POSIX threads, hem de reconèixer primer quina serà la granularitat del nostre paral·lelisme. Paral·litzarem els bucles? Les rutines que calculen les derivades?

Una unitat de paral·lelisme destaca sobre les altres: els subproblemes en els quals hem dividit el problema inicial en l'apartat 4.7. Podem dedicar un fil d'execució a cada submatriu. Aquests fils esperen (adormits, sense consumir recursos) a que es satisfacin les seves dependències, executen n iteracions seguides de la submatriu amb la qual treballen, i tornen a adormir-se, fins que es tornin a satisfer les seves dependències.

Per implementar això necessitem dues eines bàsiques del paral·lelisme: l'exclusió mutua, i condicions d'espera. POSIX Threads ens ofereix una versió ben definida de les dues eines.

- **L'exclusió mútua** és una condició que s'imposa en un codi que executen diferents fils. Aquesta condició limita l'accés al codi a un únic fil a la vegada. És a dir, si s'estan executant diversos *threads* i dos o més d'ells han d'accedir a una part del codi inclosa en les condicions d'exclusió mútua, els fils accediran d'un en un. Fins que un fil no ha sortit de la part del codi inclosa en l'exclusió mútua, cap altra pot executar aquesta part del codi. Es fa servir per llegir i actualitzar condicions de sincronització.
- **Les condicions d'espera** són una forma de sincronitzar els fils perquè s'executin en un ordre determinat. Si no es satisfà una condició, el fil s'adorm, deixant de consumir recursos de la màquina. Per despertar els processos adormits quan s'ha actualitzat la condició d'espera, es fa servir una senyal que reben tots els threads que estan esperant en la condició actualitzada. Una funció similar, anomenada **join**, fa que els fils que ja han acabat d'executar una part del codi s'esperin a que la resta del fils acabin, per unir-se una altra vegada en un únic thread i continuar l'execució de forma seqüencial.

Nosaltres creem un fil d'execució per cada submatriu existent. A més, necessitem una estructura de dades per sincronitzar els fils. Fem servir una matriu d'enters. Cada posició de la matriu correspon a una submatriu, que es troba en la mateixa

posició en la matriu general. Cada posició guarda un enter que representa la iteració a la qual ha arribat el fil corresponent.

Amb aquesta estructura és molt senzill determinar quan es pot executar un grup d'iteracions en una submatriu: quan la submatriu per l'esquerra i la que hi ha a sobre han executat la següent iteració. És a dir,

```
if(matriuCondicions[i-1][j] > matriuCondicions[i][j] &&
        matriuCondicions[i][j-1] > matriuCondicions[i][j])
                executa();
```

Amb això garantim que es respectaran les dependències entre submatrius. Malgrat això, també hem d'imposar una forta condició: que les submatrius no comencin el següent bloc d'iteracions fins que la resta de rajoles hagin acabat el bloc actual. Això és així perquè les rajoles laterals es van fent més petites. En acabar el grup d'iteracions actuals, si volem executar el següent bloc d'iteracions, necessitem la garantia que les submatrius que envolten a aquesta submatriu lateral han acabat el mateix bloc d'execucions. Però no només les immediatament adjacents, sinó en un cert radi que determina la mida inicial de la submatriu. Si no ho fem així, en executar el bloc d'iteracions següent sense que les submatrius envolvents hagin acabat el bloc anterior implicarà fer servir valors erronis.

Es pot dissenyar un mètode per detectar quan podem executar el bloc següent en cada submatriu, sense haver d'esperar a que totes acabin el bloc actual, però pot ser costós, i hem decidit deixar-ho com futura feina.

Per implementar el mètode que hem discutit aquí hem fet una rutina intermèdia en C (que té implementada la llibreria per gestionar POSIX Threads), que es cridada des del programa principal en Fortran, i que alhora crida a les rutines de càlcul de Fortran.

5.1.2 Mesures de rendiment

El rendiment és millor que amb el programa seqüencial. Tot i així, si la matriu inicial és de mida mitjana, es nota que el sistema s'ofega i creix el temps de sistema, amb molts threads esperant per ser executats. L'únic temps pitjor és en el cas del problema gran en el servidor, desactivant la precisió amb reals:

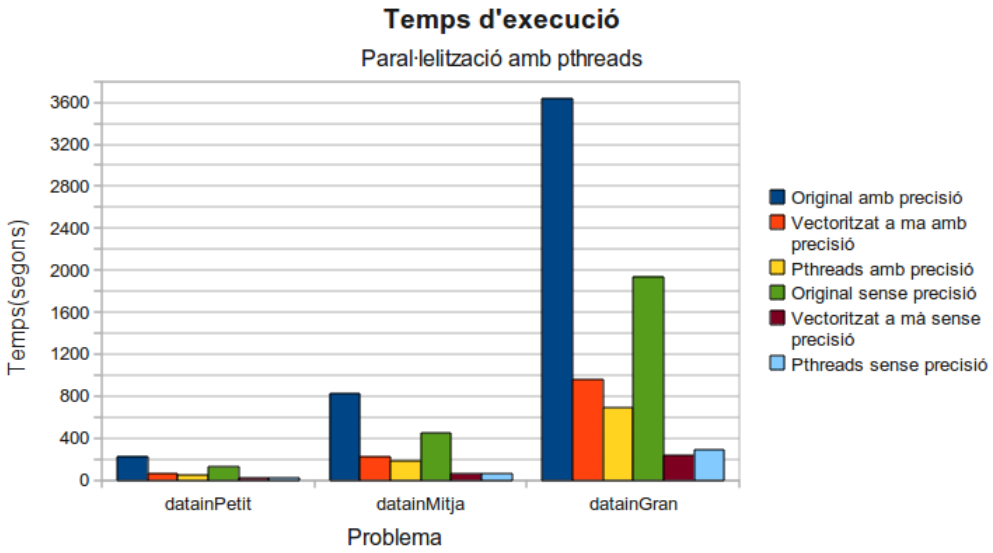


Figura 5.1: *Comparació de versions seqüencials i paral·lelitzada, executades al servidor*

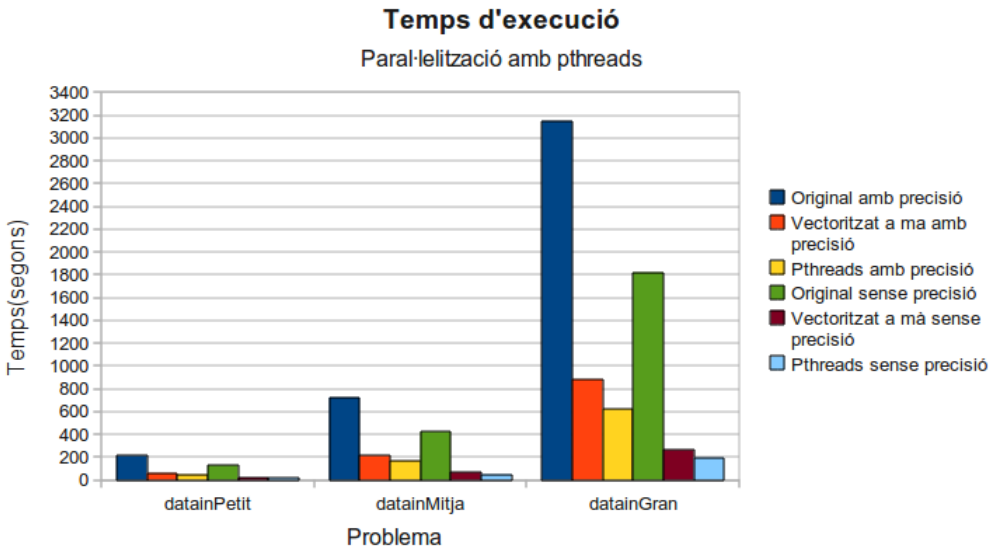


Figura 5.2: *Comparació de versions seqüencials i paral·lelitzada, executades al portàtil*

5.2 Paral·lelització amb OpenMP

OpenMP és una altra interfície de programació d'aplicacions (API) per paral·lelitzar programes que s'executen en sistemes amb memòria compartida. Aquesta interfície fa molt senzilla la paral·lelització de fragments concrets de codi, com per exemple bucles amb alts costos d'execució, o fragments independents de programa.

OpenMP està estandarditzat, i soportat per múltiples compiladors i llenguatges, com és el cas de Fortran, C/C++, i els compiladors que fem servir en el nostre projecte: gcc/gfortran i icc/ifort. Per fer-lo servir només cal introduir unes directives en el programa i compilar-lo amb una opció especial. Així, si el mateix codi es compila en una màquina que no dóna suport a OpenMP, el codi funciona igualment (sols que s'executa en sèrie)

Per paral·lelitzar el codi, OpenMP ens ofereix diferents maneres de dividir l'execució en diferents fils:

- A nivell de bucle (**OpenMP-for**): en arribar a un bucle, el fil principal es divideix en diversos fils, cadascun dels quals executa una o més iteracions del bucle.
- A nivell de secció (**OpenMP-section**): en arribar a la directiva, el fil principal es divideix en fils, cadascun dels quals executa una secció del codi compresa entre unes directives secundàries que indiquen la divisió exacta del codi.
- A nivell de tasca (**OpenMP-task**): en arribar a la directiva, es designa un fil que executi la funció (tasca) especificada sota la directiva, mentres el programa continua avançant.

Nosaltres treballem a nivell de bucle i a nivell de secció. El motiu pel qual no treballem a nivell de tasca és precisament un de les desavantatges de OpenMP: no disposa d'un mecanisme explicit per sincronitzar i gestionar l'ordre d'execució de les tasques, de forma que fer quelcom semblant a l'apartat anterior i paral·lelitzar a nivell de submatriu és més complicat. No hi ha una manera senzilla de gestionar les dependències entre tasques, que són molt presents en el nostre codi, ni conseqüentment de garantir la sincronització correcta de les tasques.

Des de la UPC es va proposar afegir un sistema per detecció automàtica de dependències entre tasques, per tal de sincronitzar automàticament aquestes tasques de forma que es respectin les dependències de dades [6]. Veurem una

idea similar amb la propera interfície, SMPSs.

Per decidir quina d'aquestes opcions és la millor, farem les nostres proves amb el problema de mida mitjana, mesurarem els temps i determinarem quina és la més efectiva, tant en el servidor com en el portàtil.

5.2.1 Paral·lelització amb OpenMP-for

Per paral·lelitzar amb OpenMP els bucles, hem de trobar aquells que realitzen més feina. Diferents iteracions d'aquests bucles han de ser independents entre elles (si no, estaríem en la mateixa situació que amb les tasques que hem explicat abans).

Per sort per nosaltres, els bucles que més feina realitzen (i que més temps d'execució ocupen) són els que calculen les derivades de U i de W i dels tensors, i les seves iteracions són independents entre elles. Podem aplicar les directives d'optimització aquí i estudiar l'impacte que tenen.

En el portàtil, que té dos nuclis disponibles, els bucles repartiran les seves iteracions entre dos fils d'execució. En el servidor, que té vuit nuclis, es repartiran entre vuit fils d'execució.

A continuació oferim les mesures mitjanes amb el percentatge de processador aprofitat, en el servidor i en el portàtil.

```
Servidor:
Mitjana de temps: 166.8 s
Mitjana d'ús de CPUs: 622% (màxim 800%)
```

```
Portàtil:
Mitjana de temps: 130.5 s
Mitjana d'ús de CPUs: 194% (màxim 200%)
```

5.2.2 Paral·leltització amb OpenMP-sections

El principi que hi ha darrera de la directiva OpenMP-sections és la de trobar fragments de codi que siguin independents entre ells, per executar-los en paral·lel. Com abans, hem de trobar parts del nostre programa que realitzin molta feina, independents entre elles, i aplicar la directiva en aquestes seccions.

Les rutines que calculen les derivades són les candidates ideals, ja que acumulen la major part del temps d'execució. Si reordenem una mica el codi, serem capaços de dividir-lo en parts independents. Per exemple, en la rutina que calcula els tensors, podem veure que el càlcul dels tensors τ_{xx} i τ_{zz} és independent del càlcul del tensor τ_{xz}. Aplicarem la directiva aquí.

Anàlogament, en la rutina que calcula U_{new} i W_{new} a partir de les derivades dels tensors, podem dividir el codi en dues parts: la que calcula U_{new} i la que calcula W_{new}.

En el portàtil, cada secció serà executada per un nucli. En el servidor, si no limitem el nombre de fils a dos (un per secció), cada secció s'intentarà executar amb més d'un fil, la qual cosa farà que creixi el temps de sistema i el programa tardi més. Per això limitem el nombre de fils a 2. De totes formes, veiem que aquest mètode no treu el màxim profit dels recursos del servidor: tindrem 6 nuclis ociosos.

Les mesures obtingues són:

```
Servidor:
Mitjana de temps: 130 s
Mitjana d'ús de CPUs: 199% (màxim 200%) --> Limitant a 2 fils
```

```
Portàtil:
Mitjana de temps: 120 s
Mitjana d'ús de CPUs: 196% (màxim 200%)
```

5.2.3 Paral·lelització amb OpenMP-sections i OpenMP-for

No hi ha res que ens limiti a fer servir només una de les tècniques explicades en les seccions anteriors: podem combinar les dues directives per realitzar els dos tipus de paral·lelisme. Aquesta combinació només té un sentit real en una màquina que pugui tractar més de dos fils d'execució alhora, com per exemple el servidor, ja que dividirem el codi primer en dues seccions, i per cada secció, en més fils dintre dels bucles.

Amb els vuit nuclis del servidor, decidim fer una distribució equitativa, designant un fil per cada secció. Aquests fils es replicaran en tres més, per paral·lelitzar

els bucles de cada secció, fent un total de vuit threads simultanis com a molt. En el portàtil, el comportament és idèntic al de OpenMP-sections, ja que com que només disposem de dos nuclis, no executa els bucles en paral·lel, només les seccions.

```
Servidor:
Mitjana de temps: 177.33 s
Mitjana d'ús de CPUs: 658% (màxim 800%)
```

```
Portàtil:
Mitjana de temps: 119 s
Mitjana d'ús de CPUs: 199% (màxim 200%)
```

5.2.4 Conclusions i mesures

De totes les mesures anteriors, podem concloure que la millor versió en el portàtil és, sense dubte, OpenMP-sections.

Pel servidor, tot i que aquesta OpenMP-sections és la millor versió en mitjana, no és la més desitjable: hem d'aprofitar més les unitats de càlcul que tenim disponibles. Més endavant veurem com treure més profit amb una altra interfície. De fet, estudiant els resultats amb qualsevol de les altres versions, podem deduir que si s'aprofités millor el processador (apropant-se al 800% màxim de rendiment), els temps serien més petits que la versió amb OpenMP-sections. L'ús de la CPU sembla augmentar quan més gran és el problema. De fet, amb l'experiment de mida gran, els temps són menors en mitjana amb la versió OpenMP-for+sections:

```
Servidor amb problema gran i OpenMP-sections:
Mitjana de temps: 757 s
Mitjana d'ús de CPUs: 199% (màxim 200%)

Servidor amb problema gran i OpenMP-for+sections:
Mitjana de temps: 464 s
Mitjana d'ús de CPUs: 750% (màxim 800%)
```

Aquesta variabilitat en l'aprofitament de la CPU entre execucions i problemes fa que els resultats siguin molt més dispersos. Oferim una gràfica amb mitjanes calculades amb 4 mesures, en comptes de 3, però advertim que la desviació estàndard és més gran que en la resta de gràfiques. Aquests són el resultats sempre amb precisió dels reals, ja que si desactivem, els temps es fan més petits i resulta difícil comparar les mesures:

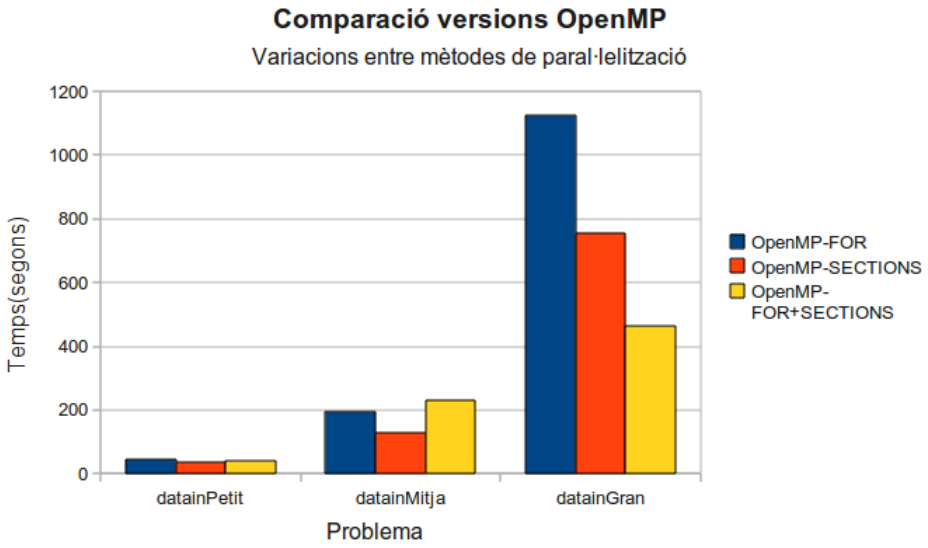


Figura 5.3: *Comparació de versions d'OpenMP al servidor*

El nostre objectiu, llavors, serà aprofitar millor la resta de nuclis i reduir encara més els temps de la versió en el servidor.

Per comparar entre versions de interfícies de paral·lelització, ens quedem amb OpenMP-sections en ambdós casos, però sense oblidar que no treiem tot el rendiment possible en el servidor.

5.3 Paral·lelització amb SMPSs

SMPSs (Symmetric Multiprocessors Superscalar) és una interfície dissenyada en el Barcelona Supercomputing Center per paral·lelitzar programes de forma senzilla, flexible i portable. L'objectiu de SMPSs és explotar de manera automàtica, amb el mínim canvi possible per al programador, les arquitectures amb diversos nuclis.

Funciona, al igual que OpenMP, amb directives inserides directament al codi, que indiquen a un compilador codi a codi (això és, un compilador que, en comptes de generar llenguatge màquina, genera codi d'alt nivell), quines rutines es poden paral·lelitzar. El compilador SMPSs genera un graf acíclic dirigit que representa les dependències funcionals de dades entre rutines. D'aquesta forma, es poden paral·lelitzar l'execució d'aquests mètodes, sincronitzant-los per respectar

les dependències de dades.

Per declarar una rutina com a paral·lelitzable només cal afegir una directiva, especificant quins paràmetres són únicament d'entrada, quins són de sortida i quins d'entrada i sortida. La idea bàsica és que una tasca esperarà a que qualsevol tasca que tingui com a paràmetre de sortida o d'entrada-sortida un dels seus paràmetres d'entrada s'acabi d'executar abans d'executar-se ella mateixa.

5.3.1 Implementació

En una aplicació on les dependències de dades són tan complexes com les nostres (amb submatrius que es desplacen per una matriu més gran, algunes d'elles fent-se més petites, i a sobre iterant tot el sistema), no basta amb indicar quines rutines són paral·lelitzables i el sentit dels seus paràmetres. Si ens limitem a dir que la rutina que tracta una submatriu és paral·lelitzable, la interfície, al no poder detectar de quines dades depèn exactament, serial·litzarà tot el sistema i no guanyarem res.

Per ajudar a la sincronització, fem servir una tècnica similar a la que hem aplicat amb POSIX Threads: una matriu de sentinelles. Aquesta matriu té una posició per cada submatriu del problema, i cada posició és simplement un enter.

Com a eina de sincronització, fem servir una rutina que rep paràmetres però no realitza cap operació. La seva estructura en C és:

```
#pragma css task input ( sentinel_1, sentinel_2 ) inout( sentinel_3 )
void accum ( int * sentinel_1, int * sentinel_2, int* sentinel_3  )
{}
```

Quin sentit té aquesta rutina que no fa res? Si la declarem com a paral·lelitzable (com indica la directiva `#pragma css task`), i indiquem de la manera especificada els seus paràmetres, la rutina actuarà com a mètode per sincronitzar les tasques. La idea és senzilla: les rutines que tenen els sentinelles 1 i 2 (`sentinel_1` i `sentinel_2`) com a paràmetre de sortida es poden executar en paral·lel. Les rutines que tenen com a paràmetre d'entrada el sentinella 3 (`sentinel_3`), han d'esperar a que les rutines que fan servir els sentinelles 1 i 2 s'acabin d'executar.

Esquemàticament:

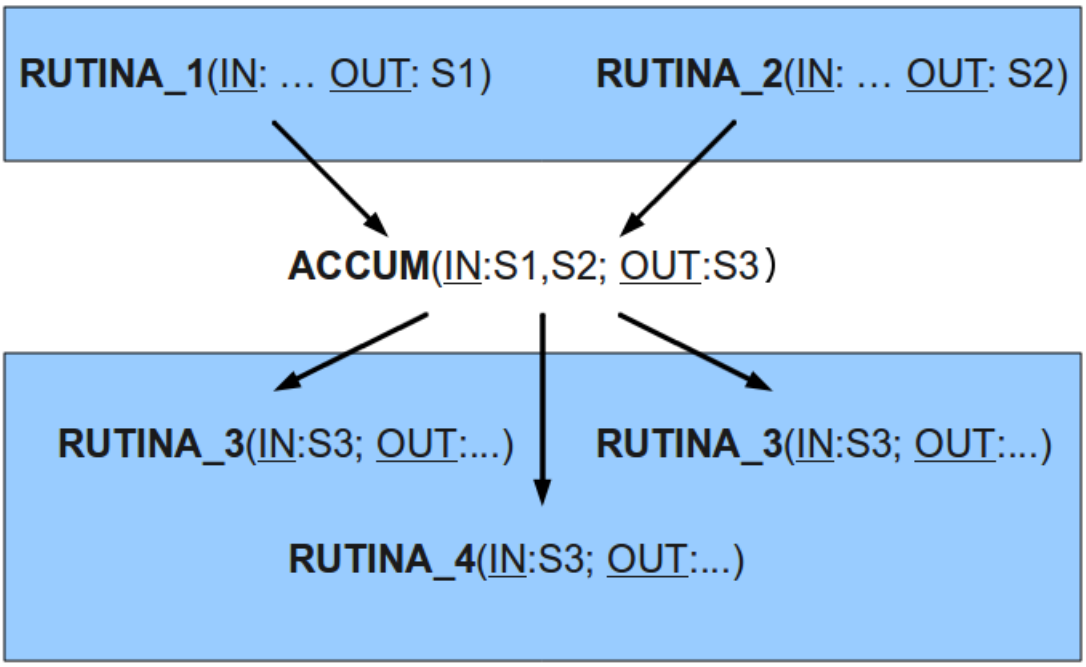


Figura 5.4: *Sincronització de les rutines amb SMPSs. En blau, rutines que es poden executar en paral·lel.*

Amb aquesta rutina, la sincronització és senzilla. Declarem la rutina que executa les iteracions d'una submatriu com a paral·lelitzable amb la directiva. Aquesta rutina té com a paràmetre de sortida el sentinella que ocupa la posició equivalent a la submatriu en la matriu de sentinelles.

Abans de la crida a la rutina, fem una crida a la rutina *accum*. Com a sentinelles d'entrada, passem el que equival a la submatriu esquerra i el que equival a la submatriu superior de la submatriu actual. Així, garantim que quan s'executi la submatriu actual, les seves dependències (les dades de la submatriu adjacent esquerra i superior) ja estaran satisfetes.

```
accum(&sentinelles[i-1][j], &sentinelles[i][j-1], &sentinelles[i][j]);
tile(...,&sentinelles[i][j]);
```

Per les matrius lateral esquerra o superior fem servir un sentinella buit per la dependència que falta (perquè es troba en un costat). A més, com que les matrius superiors (les de la superfície lliure) tenen una mida diferent a la resta de submatrius, a vegades és necessari que activin dos sentinelles, ja que no encai-

xen perfectament (i una submatriu superior pot englobar a més d'una submatriu inferior):

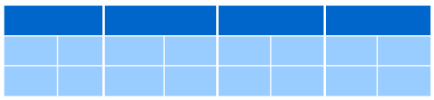


Figura 5.5: *Exemple del desajust i la necessitat d'actualitzar dos sentinelles.*

Tot i així, es manté la condició que evita que iniciem un altre bloc d'iteracions amb les submatrius si encara queden submatrius que no han calculat el bloc anterior. Això ho implementem amb una directiva de SMPSs que evita que els fils avancin fins que no arribin tots al punt del codi on és la directiva.

5.3.2 Mesures

Un com implementat el paral·lelisme amb SMPSs, hem de fer proves i valorar l'escalabilitat del programa per determinar la millor versió.

Per poder gestionar el nombre de fils d'execució que farà servir l'aplicació, SMPSs ens ofereix una variable d'entorn, 'CSS_NUM_CPUS', on podem indicar el nombre de fils d'execució que volem actius. Si posem més fils que nuclis té la nostra màquina, els temps creixen molt, principalment pel temps de sistema en el qual el ordinador gestiona els múltiples threads per executar-los.

En el portàtil no té massa sentit parlar d'escalabilitat: amb només dos nuclis, tenim tres opcions: o indicar només un fil d'execució, i llavors ens trobem en el cas seqüencial; dos fils d'execució, que permet paral·lelisme de l'aplicació; o més de dos fils, que es contraproduent perquè ens dóna pitjors temps.

En el servidor tenim fins a 8 fils disponibles. Els aprofita bé el programa? Fem una prova, amb el problema de mida mitjana i demanant precisió en les operacions. Cada valor de 'CSS_NUM_CPUS' s'executa 5 vegades:

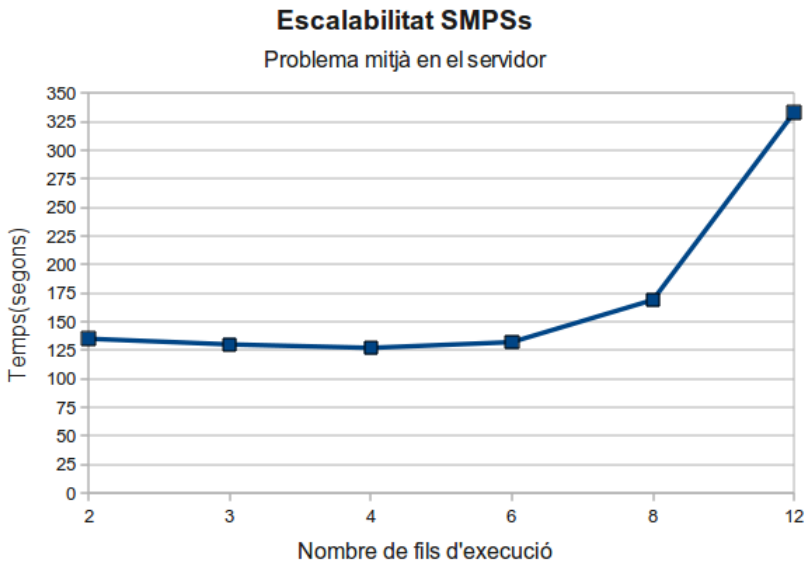


Figura 5.6: *Gràfica que il·lustra l'escalabilitat de l'aplicació amb SMPSs al servidor*

Els millors resultats els obtenim amb 4 fils d'execució: entorn a 2 min i 6 segons. Estem, però, en el mateix cas que abans. No aprofitem tots els recursos que ens dona la màquina: la ocupació del processador és d'un 400%, quan el màxim teòric és del 800% (perquè tenim 8 nuclis). No hi ha manera de treure més profit al servidor?

5.4 Combinació de SMPSs i OpenMP

La solució passa per combinar dues de les interfícies de paral·lelisme per explotar més els recursos de la màquina. Aquestes dues interfícies es complementen perfectament: SMPSs genera paral·lelisme a nivell de subrutina, explota la divisió en subproblemes que hem aplicat al nostre codi; OpenMP aplica un paral·lelisme a nivell intern de la rutina: ens ajuda a reduir els temps dels bucles o les seccions independents.

Què succeeix si combinem les dues interfícies? Ho veurem fent proves en el servidor. En el portàtil, aquesta combinació no té massa sentit, ja que només disposem de dos nuclis per paral·lelitzar: si sobrecarreguem la màquina, els temps empitjoraran. En el servidor, hem de vigilar no excedir el nombre de fils disponibles.

Si, per exemple, escollim 4 fils per executar en paral·lel les submatrius amb SMPSs, llavors podem paral·lelitzar les rutines que calculen les derivades amb OpenMP-for o OpenMP-sections amb dos fils (4 fils per executar submatrius x 2 fils per calcular derivades en cada submatriu $= 8$ fils màxim a la vegada).

En la següent gràfica es pot veure el resultat, seleccionant 4 fils per executar submatrius, i 2 fils per OpenMP-for o OpenMP-sections. Es posa també com a exemple OpenMP-for+sections, amb un total de (4 fils per les submatrius x 2 fils per les seccions x 4 fils pels bucles=) 32 fils, per veure que la màquina es sobrecarrega i augmenta el temps. El programa s'executa amb el problema de mida mitjana, i les proves es repeteixen 5 vegades.

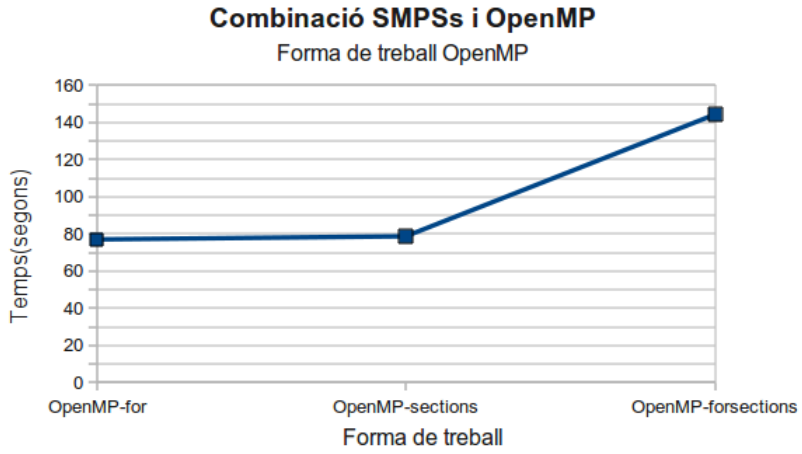


Figura 5.7: *OpenMP-for i OpenMP-sections es comporten igual de bé*

Tot i que els temps són molt similars, ens quedem amb OpenMP-for, que té un temps lleugerament menor.

5.5 Solapament d'iteracions

El no poder començar a executar el següent bloc d'iteracions fins que totes les submatrius no han acabat el bloc actual és una important limitació per la paral·lelització del codi.

Dissenyar una rutina per detectar quan es pot executar el següent bloc sense esperar a la resta, i que no consumeixi molt temps d'execució és una feina delicada, ja que és fàcil cometre errors i el codi és molt difícil de corregir per l'execució en paral·lel i la mida del programa.

La següent modificació que proposem soluciona aquesta situació en part. Tot i que aconseguim una reducció significativa del temps d'execució, aquesta optimització només es pot fer si es coneix una mica com funciona el codi, i si se sap la mida del problema i el nombre de submatrius en el que es dividirà abans de compilar.

La idea és senzilla: fer que la primera submatriu, que correspon a la cantonada superior esquerra, depengui d'un sentinella que s'actualitza en una de les últimes submatrius (però no l'última, o el profit que li traurem serà nul). Quan la submatriu en qüestió actualitzi aquest sentinella, s'iniciarà l'execució del següent bloc d'iteracions.

Figura 5.8: *Quan s'executa la submatriu vermella, segur que les submatrius grogues ja s'ha executat*

Això comporta riscos: si el processador decideix donar prioritat a les submatrius del següent bloc d'iteracions abans que les que encara no s'han executat en el bloc actual, apareixeran errors en els resultats.

És per aquest motiu que només recomanem aplicar aquesta optimització en problemes d'una mida gran. Quan més gran sigui el problema i en més submatrius es pugui dividir, millor.

5.6 Comparativa final

Finalment, fem una comparativa entre les versions amb diferents interfícies.

En el portàtil, no hem implementat SMPSS i OpenMP combinats perquè no disposem de suficients nuclis per poder treure'n profit. En la versió amb SMPSS i solapació d'iteracions que hem discutit en la secció anterior no hem considerat el problema de mida petita, perquè la tècnica per solapar iteracions provoca errors massa sovint. En el cas del problema de mida mitjana, només es va produir un error en les sis proves que vam realitzar. En el problema de mida gran no es va produir cap error.

En el servidor, hem fet totes les proves, incloses les de SMPSS i OpenMP combinats. Com en el cas anterior, en la versió amb SMPSS i solapació d'iteracions no hem fet les proves amb el problema de mida petita, perquè apareixen errors massa sovint. En la resta de problemes no vam detectar cap error.

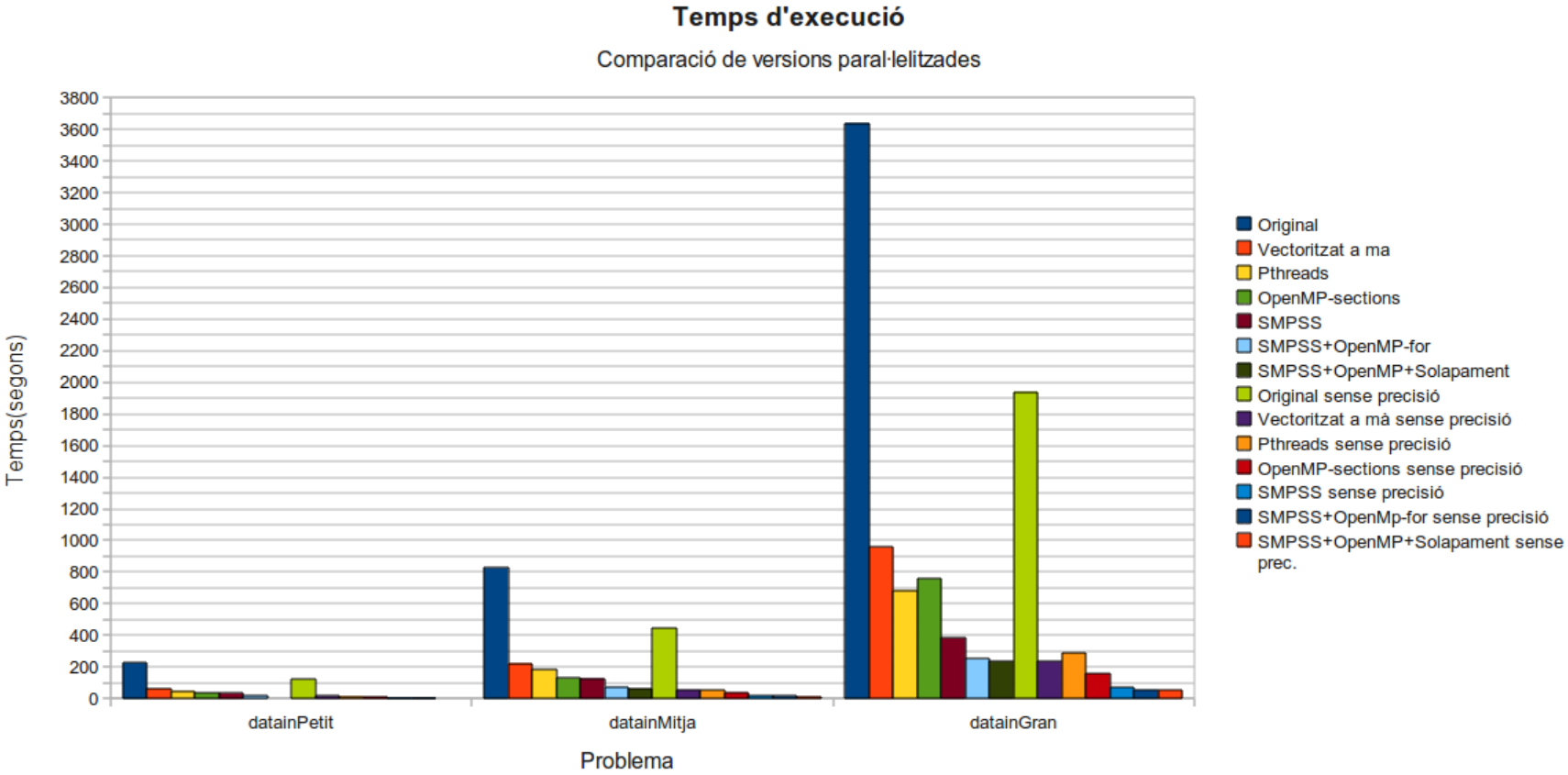


Figura 5.9: *Comparació de versions paral·lelitzades al servidor.*

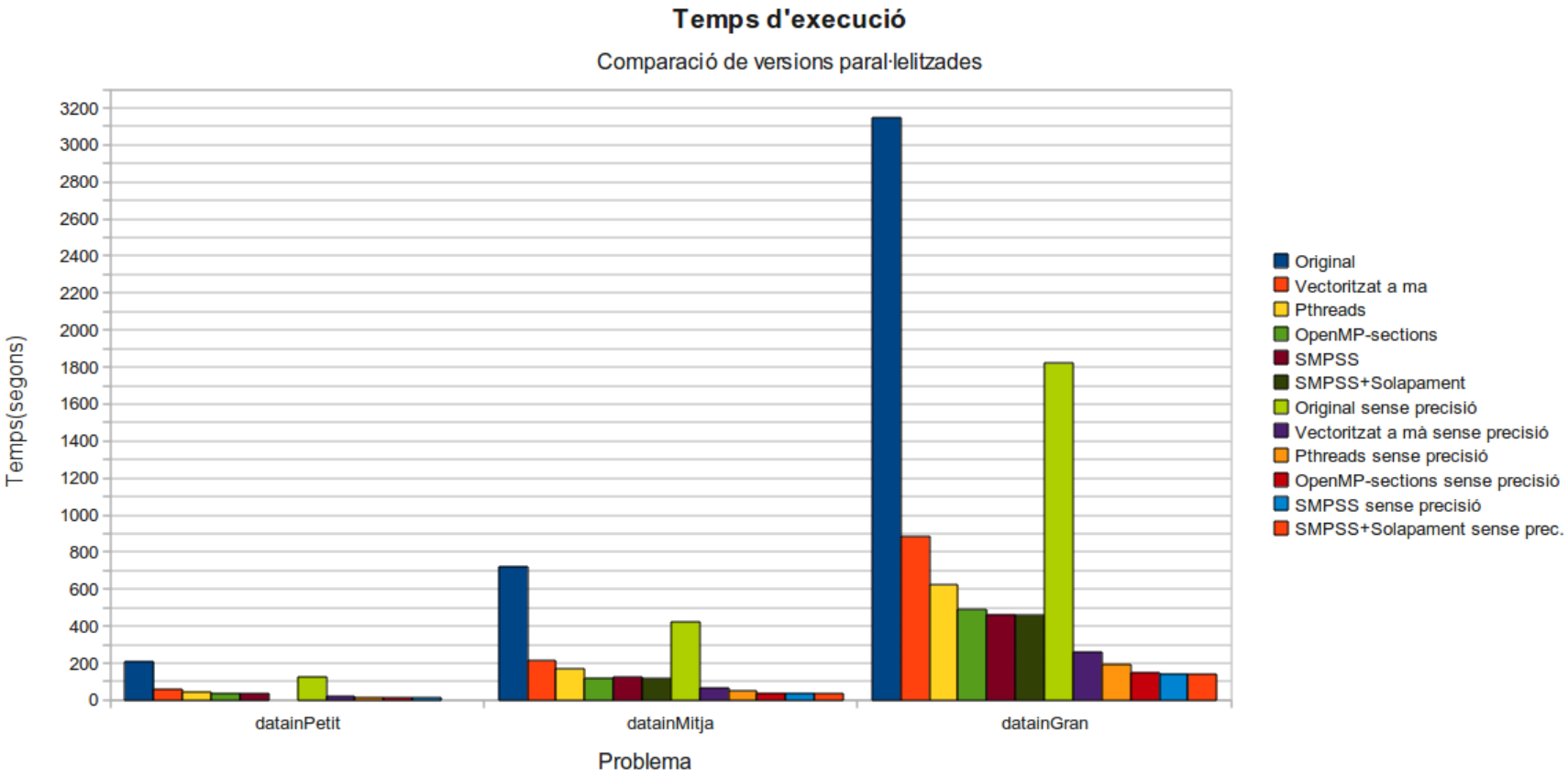


Figura 5.10: *Comparació de versions paral·lelitzades al portàtil.*

6

Conclusions

6.1 Mesures finals

És el moment de fer un resum de tot el que hem aconseguit. Primer de tot, analitzem els millors temps:

Servidor			
	Original II	Original III	Millor versió
datain petit	406 s	226 s	20 s
datain mitjà	1422 s	826 s	63 s
datain gran	6183 s	3640 s	235 s
Portàtil			
	Original II	Original III	Millor versió
datain petit	524 s	210 s	35 s
datain mitjà	1818 s	723 s	116 s
datain gran	7668 s	3148 s	458 s

Amb aquestes dades, podem donar una aproximació al factor de millora de velocitat *speed-up* que hem aconseguit al llarg del projecte. Recordem que el factor de millora de velocitat és:

$$S = \frac{Temps_{original}}{Temps_{optimitzat}}$$

Si apliquem aquesta fórmula, podem veure que els factors de millora de velocitat que hem aconseguit són:

Servidor		
	Factor respecte Original II	Factor respecte Original III
datain petit	20.3x	11.3x
datain mitjà	22.6x	13.1x
datain gran	26.3x	15.5x
Portàtil		
	Factor respecte Original II	Factor respecte Original III
datain petit	15x	6x
datain mitjà	15.7x	6.2x
datain gran	16.7x	6.9x

Com és natural, en el servidor els factors són millors perquè tenim més recursos per explotar. Podem veure la millora si les execucions es fan sense demanar valors precisos als reals. Aquests resultats són per il·lustrar que l'ordre del factor de millora és més gran, però recordem que fins avaluar els resultats contra el model analític, no podem assegurar si són prou precisos. Aquí no comparem contra gfortran, perquè els resultats que dóna sí son precisos.

Servidor, sense precisió		
	Original III	Millor versió
datain petit	122 s	6 s
datain mitjà	446 s	13 s
datain gran	1941 s	55 s
Portàtil, sense precisió		
	Original III	Millor versió
datain petit	127 s	12 s
datain mitjà	422 s	35 s
datain gran	820 s	141 s

Els factors són:

Servidor, sense precisió	
	Factor respecte Original III
datain petit	20.3x
datain mitjà	34.3x
datain gran	35.3x

Portàtil, sense precisió	
	Factor respecte Original III
datain petit	10.6x
datain mitjà	12x
datain gran	13x

En general, hem aconseguit que el programa en el servidor s'executi més de déu vegades més ràpid que el programa original, mentre que en el portàtil s'executa més de sis vegades més ràpid. Són uns resultats prou bons.

Oferim ara un diagrama per entendre com han contribuït les diferents optimitzacions a la reducció del temps del problema. Ho fem amb el cas del servidor, triant el temps inicial amb el problema gran compilat amb gfortran (i incloem la reducció deguda a l'ús de ifort). La versió és sempre amb càlculs de reals de forma precisa, ja que sense fer una comparació amb el model analític no podem garantir que els resultats arrodonits siguin correctes per la simulació.

Aproximadament, el 99% de la millora en temps d'execució es distribueix així (l'1% restant es perd en arrodoniments i variació dels resultats):

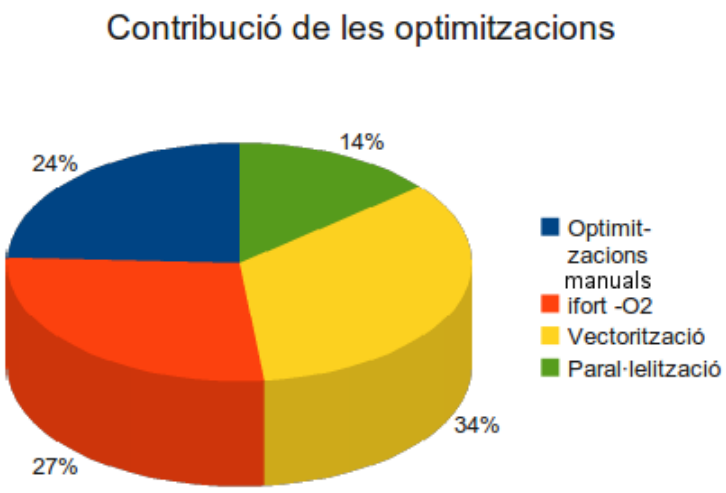


Figura 6.1: *Resum de l'impacte de les millores*

A continuació veurem l'impacte de les millores desglossat. Cal indicar que els valors d'un mateix tipus de millora són acumulats: així, per exemple, la porció que indica el guany corresponent a la vectorització a mà s'ha de sumar a la

porció que indica el guany obtingut amb l'autovectorització; la porció que indica el guany d'usar SMPSs i OpenMP conjuntament s'ha de sumar a la porció que indica el guany obtingut amb SMPSs, etc.

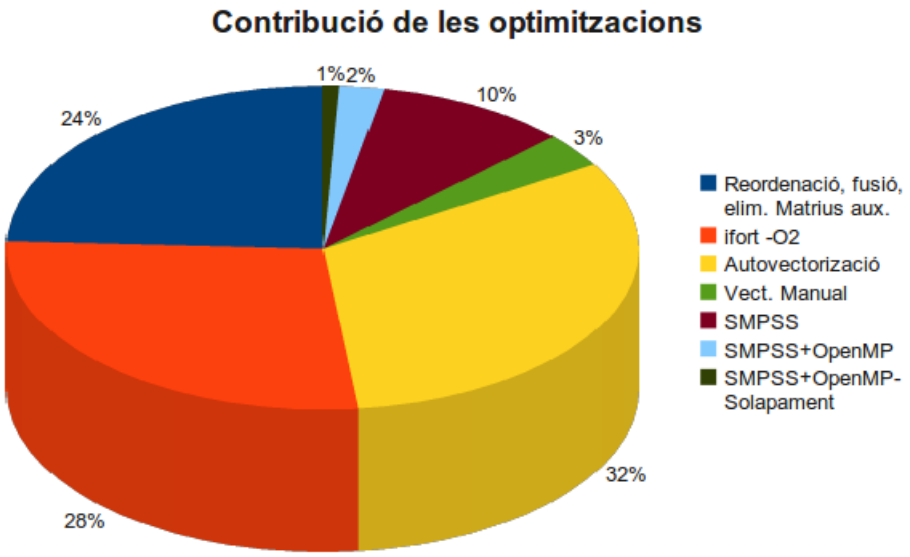


Figura 6.2: *Resum desglossat de l'impacte de les millores*

Per acabar, executarem el programa amb el problema més gran: les malles tenen 4819x4819 elements, i la simulació es realitza durant 17 segons de temps del problema. Amb el resultat de les mesures, volem posar de manifest la importancia de les optimitzacions.

El **programa original**, executat amb el compilador d'Intel (que ja realitza algunes optimitzacions i vectorització automàtica) al servidor, tarda **21 h i 26 min** en executar-se, o el que és el mateix, 77160 segons. El mateix problema, executat amb **la millor versió del nostre programa** (fent servir SMPSs i OpenMP amb iteracions solapades), tarda **54 min i 35 segons**, és a dir, 3275 segons. El nostre programa, en aquesta situació, s'ha executat 23.6 vegades més ràpid que l'original (23.6x).

En el temps en el que es fa una simulació amb el programa original, nosaltres podriem fer més de 21 simulacions d'un problema de mida similar. Això ens permetria obtenir més resultats dels experiments i extreure'n conclusions de forma molt més ràpida. O fer experiments amb moltes més dades.

És indiscutible que optimitzar els programes perquè s'executin més ràpidament ofereix moltes avantatges.

6.2 Treball futur

El projecte ens ha obert algunes portes per continuar treballant per tal de reduir més el temps d'execució i valorar el rendiment de les aplicacions que hem comparat en altres sistemes.

- **Comparar els resultats arrodonits amb el model analític**: per determinar si els resultats que ens ofereix el compilador d'Intel, quan permitim que faci operacions amb reals arrodonint, són acceptables per la nostra simulació, hem de comparar aquests resultats amb els que ens dona el model analític original, mesurar la discrepància i decidir. Si acceptem aquests valors, el temps d'execució es redueix molt, ja que el compilador d'Intel crea codi que treballa molt més ràpid amb els nostres reals.

- **Paral·lelitzar el programa per un sistema distribuït o una targeta gràfica**: el programa, tal i com l'hem dissenyat nosaltres, és més efectiu en un sistema amb memòria compartida, on tots els processadors (i fils d'execució) tenen accés a les mateixes dades. Seria interessant dissenyar la paral·lelització en un entorn en el que els fils no poden compartir tanta informació (perquè es ralentitza l'execució si s'ha de transmetre molta informació), com un sistema distribuït o una targeta gràfica, amb la interfície CUDA de nVidia, per exemple. Un exemple d'aquest sistema el podem trobar a [8].

- **Millorar el solapament d'iteracions**: tal i com hem dissenyat nosaltres, els blocs d'iteracions es poden solapar, però si la malla amb la qual treballem no és prou gran, poden aparèixer errors amb facilitat. Fins i tot si és molt gran no podem garantir que l'execució sempre sempre serà correcta. És necessari dissenyar una rutina per detectar quan es pot executar el següent bloc d'iteracions sense esperar a que totes les submatrius acabin l'actual

- **Provar la simulació en entorns amb més processadors**: per veure com escala el temps d'execució amb més processadors, es pot provar l'execució en entorns amb molts processadors o en un supercomputador.

6.3 Valoració personal

Vaig seleccionar aquest projecte entre les ofertes disponibles a la facultat perquè tractava dos dels temes que més m'interessen de l'Enginyeria Informàtica: l'optimització i la paral·lelització de programes. Essent la computació científica i d'alt rendiment un dels aspectes que em va atreure per estudiar Enginyeria Informàtica en primer lloc, i veient la tendencia actual a l'hora de dissenyar ordinadors, crec que aquests dos temes, l'optimització i la paral·lelització, són indispensables per desenvolupar amb èxit bones aplicacions d'anàlisi numéric o simulació.

Durant els meus estudis vaig tenir l'oportunitat de cursar assignatures sobre el tema (PCA, Programació Conscient de l'Arquitectura; PCD, Programació Concurrent i Distribuïda, etc.) i ja estava familiaritzat amb els processos que hi ha darrera.

Un altre gran al·licient per seleccionar aquest projecte va ser la possibilitat de combinar-lo amb un tema que també m'interessa molt: la Física, i el fet que la informàtica s'ha convertit en un dels pilars fonamentals per l'estudi d'aquesta ciència. Els meus directors de projecte, en saber que també estudio Física, em van proporcionar un programa per optimitzar que és una simulació geofísica real, que es pot utilitzar i que té unes bases matemàtiques que el fonamenten.

Durant èpoques de treball en el projecte, a vegades ho he passat malament quan cometia errors i no sabia per on continuar. Els meus directors m'han ajudat en aquests moments i moltes altres vegades, i en general, passat els moments difícils, he gaudit treballant en això. Els resultats crec que valen la pena. I el que més valoro és que he aprés molt de moltes coses: he aprés un llenguatge nou, com és Fortran, molt utilitzat encara en la comunitat científica; he conegut dues interfícies de paral·lelització noves i molt útils, OpenMP i SMPSs; he aprés a complir una metodologia de treball i organització (cal ser molt organitzat quan treballes re-dissenyant un programa, i guardant versions intermèdies, o et perds ràpidament); he pogut posar en pràctica molts coneixements adquirits al llarg de la carrera; etc.

Per tot això, la meva valoració personal sobre tot el projecte és molt positiva.

7

Desenvolupament del projecte. Memòria econòmica

En aquesta secció resumirem el procés de desenvolupament del nostre projecte, explicant les etapes principals del mateix, i presentarem un informe econòmic il·lustratiu que reflectirà el cost d'aquest procés.

7.1 Metodologia de disseny

Com ja hem explicat breument a la introducció, la nostra metodologia de disseny té les característiques del procés de disseny iteratiu i incremental. Tot i que no estem produint software nou, en el sentit estricte de la paraula, si estem re-dissenyant un software existent i afegint canvis que, tot i que no generen noves funcionalitats del sistema, modifiquen el comportament intern d'aquest per satisfer unes necessitats (bàsicament, reduir el temps d'execució i augmentar el rendiment).

El procés de disseny iteratiu i incremental es basa en produir el software per etapes. En cada etapa es fa una anàlisi dels requisits i del problema, s'elabora una solució per a aquest problema, s'implementa aquesta solució, es prova per verificar que es correcta, i s'avalua per determinar si soluciona correctament el problema detectat en la fase d'anàlisi. Si és el cas, llavors s'incorpora el canvi al sistema i s'avança al següent problema.

En el nostre procés de disseny, l'anàlisi és la detecció dels punts crítics en l'execució del programa; el disseny i implementació de la solució és introduir els

canvis en el codi que milloren el rendiment; en la fase de prova ens assegurem que els canvis introduïts no tinguin cap error, i, finalment, la fase d'avaluació consisteix en mesurar l'impacte de l'optimització.

El disseny iteratiu presenta avantatges respecte a altres processos de disseny, com per exemple el disseny en cascada. Un d'aquests avantatges és que es triga molt poc en determinar si un canvi és positiu per al sistema. Aquesta qualitat és quelcom molt desitjable en el procés d'optimització d'un programa: la dificultat de considerar tots els recursos dels quals disposa la màquina i tots els factors que afecten al rendiment d'un algorisme fa que, a vegades, un canvi que ens sembla pot millorar molt el rendiment d'un programa pot no només no millorar-lo, sinó empitjorar aquest rendiment. Imaginem que treballem en un procés de disseny en cascada. Mesos planificant canvis, re-dissenyant el codi, fent proves per comprovar que tot funciona correctament, i en el moment d'avaluar la millora, ens adonem que el temps ha empitjorat. Mesos de feina per res.

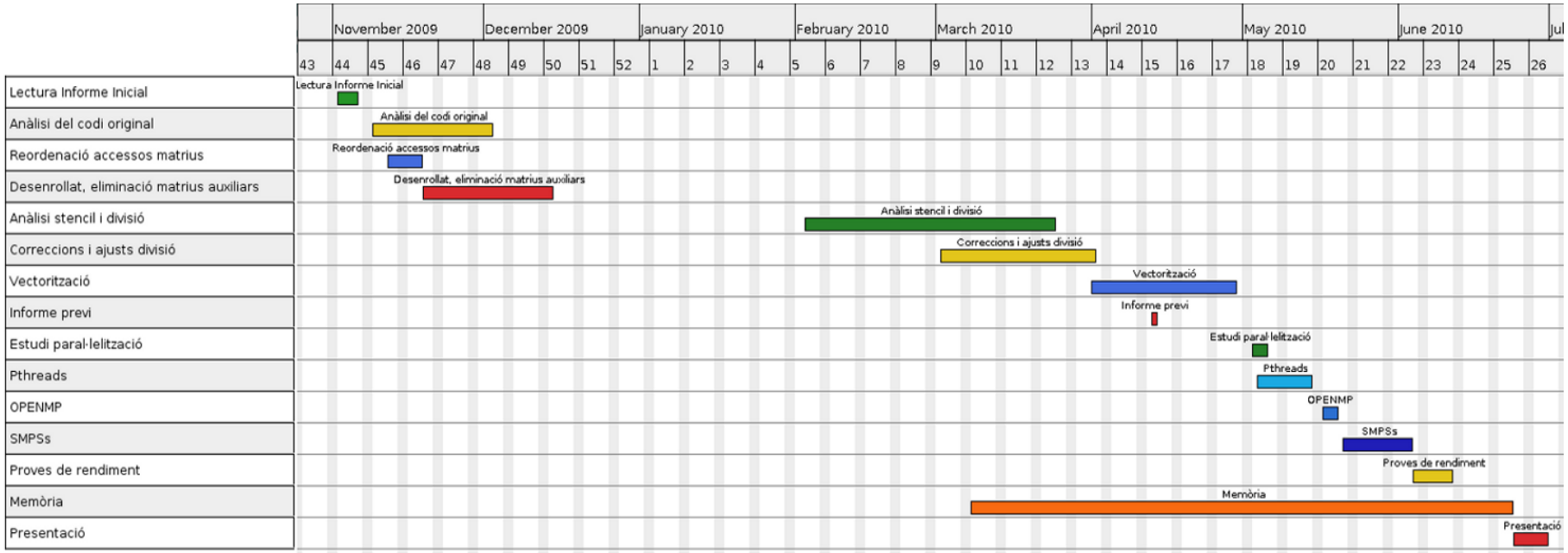
Tot i així, aquesta metodologia de disseny també presenta desavantatges. El més clar és la manca d'una visió global de tot el procés d'optimització. Tots els canvis i millores es fan respecte a l'etapa anterior, o com a molt, dos o tres etapes anteriors. Això fa que sigui més fàcil passar per alt un punt clau en el procés de disseny, que ens hagués portat a una millor optimització del programa.

Podem fer una analogia amb el mètode de programació amb algorismes voraços (*greedy algorithm*): en aquest mètode de programació, en cada etapa del procés de l'algorisme, s'agafa la millor opció segons les condicions actuals del problema, en comptes de la situació global. Aquesta tècnica generalment ens porta a una solució bona, però sovint aquesta bona solució no és la millor de les possibles.

Amb el procés de disseny iteratiu passa quelcom similar: apliquem millores que ens donen un rendiment més alt que el que teníem fins ara, però sovint hem de prendre decisions que ens porten cap a una direcció o una altra, i la major part de les vegades, no tornem enrere per visitar altres camins: ens quedem amb la millor solució que hem anat formant. Això no vol dir que sigui la millor.

7.2 Diagrama de planificació

Aportem un diagrama amb la planificació del projecte:


November 2009
December 2009
January 2010
February 2010
March 2010
April 2010
May 2010
June 2010
43 44 45 46 47 48 49 50 51 52 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26
Lectura Informe Inicial
Anàlisi del codi original
Reordenació accessos matrius
Desenrollat, eliminació matrius auxiliars
Anàlisi stencil i divisió
Correccions i ajusts divisió
Vectorització
Informe previ
Estudi paral·lelització
Pthreads
OPENMP
SMPSs
Proves de rendiment
Memòria
Presentació

7.3 Pressupost

En aquesta part discutirem el cost econòmic del projecte.

7.3.1 Cost del treball

A continuació detallem el cost estimat del projecte. Aquest cost és una aproximació que s'ha calculat tenint en compte el temps dedicat a la lectura de documentació, a l'anàlisi i disseny de les optimitzacions, a les proves i la mesura de rendiment, i finalment, a la redacció de la memòria.

El temps, amb les tasques desglossades:

Hores dedicades segons tasca	
Tasca	**Temps**
Lectura de documentació	15 h
Anàlisi: - *Programa* - *Requisits* - *Eines*	60 h
Programació: - *Optimitzacions* - *Paral·lelització* - *Exemples*	200 h
Proves: - *Correcció d'errors* - *Configuració eines*	125 h
Avaluació i mesures:	75 h
Documentació:	130 h
Total:	**605 h**

Si suposem un hipotètic sou d'un enginyer informàtic que treballa per hores, a 35 €/hora, el cost total de les hores de feina per optimitzar el codi puja a:

Cost del projecte: 21175 €

7.3.2 Despeses d'eines i material

Durant tot el nostre projecte, hem necessitat una sèrie d'equip i eines que tenen un cost econòmic. Per tal de reflectir de forma més acurada el cost total del projecte, és necessari incloure el preu d'aquestes eines materials en el pressupost.

- **Les màquines**: per provar el codi, hem necessitat dues màquines amb components diferents per tal de tenir una visió més amplia del comportament del codi en diferents entorns. Malgrat només hem fet servir dues màquines diferents, seria desitjable disposar de més màquines, especialment màquines amb diferent nombre de nuclis, per apreciar l'escalabilitat de la paral·lelització amb més o menys fils disponibles. No obstant això, les màquines comuns actuals tenen entre 1 i 8 processadors. Si volem més quantitat, hem de recórrer a supercomputadors o sistemes distribuïts.

  Les màquines que hem fet servir són dos: un portàtil, on es desenvolupaven els canvis, i un servidor, l'arquitectura del qual era a la que es dirigien les optimitzacions.

  - Portàtil: Dell Studio XPS 13 amb Intel Core 2 Duo P9600 (2.66 GHz, FSB 1066 MHz, 6 MB cache), amb 4 GB de RAM (1067 MHz, DDR3), disc dur de 500 GB i targeta gràfica nVidia 9500m de 256 MB. Total: 1228 €
  - Servidor: Dual Quad Core Intel Xeon E5520 (2.27 GHz, 8 MB cache) amb 8 GB de RAM. Total: 1900 €aprox.

- **Llicències**: tot i que hem treballat amb les versions gratuïtes d'avaluació, és recomanable disposar de llicencies completes dels compiladors d'Intel, **ifort** i **icc**. La llicència de icc per Linux val $599, mentre que la de ifort val $899. Com que per obtenir la millor versió fem servir els dos compiladors, es pot adquirir la *Intel Compiler Suite Professional Edition*, que inclou els dos compiladors, per $1349

Annex A

Fitxers d'entrada dels experiments

A.1 Mida petita

```
899                     nx      (GRID)
899                     nz
9000.                   lx      (DOMAIN space - time)
9000.                   lz
3.                      Tfinal
11025000000.            lambda  (Lame Parameters & density)
11025000000.            miu
2500.                   rho
500.                    alpha   (Source parameters)
0.25                    rise
300             X index of source location
0               Z index of source location
400             X index of 1st receiver
0               Z index of 1st receiver
500             X index of 2nd receiver
0               Z index of 2nd receiver
600             X index of 3rd receiver
0               Z index of 3rd receiver
700             X index of 4th receiver
0               Z index of 4th receiver
800             X index of 5th receiver
0               Z index of 5th receiver
850             X index of 6th receiver
0               Z index of 6th receiver
850             X index of 7th receiver
2               Z index of 7th receiver
500             X index of 8th receiver
2               Z index of 8th receiver
700             X index of 9th receiver
2               Z index of 9th receiver
750             X index of 10th receiver
```

```
0               Z index of 10th receiver
0.95            Maximum damping to Cerjain's tappering
50              number of grid lines for the Cerjain's tappering
```

A.2 Mida mitjana

```
1399                    nx      (GRID)
1399                    nz
14000.                  lx      (DOMAIN space - time)
14000.                  lz
4.                      Tfinal
11025000000.            lambda  (Lame Parameters & density)
11025000000.            miu
2500.                   rho
500.                    alpha   (Source parameters)
0.25                    rise
750             X index of source location
0               Z index of source location
850             X index of 1st receiver
0               Z index of 1st receiver
900             X index of 2nd receiver
0               Z index of 2nd receiver
1000            X index of 3rd receiver
0               Z index of 3rd receiver
1100            X index of 4th receiver
0               Z index of 4th receiver
1200            X index of 5th receiver
0               Z index of 5th receiver
1300            X index of 6th receiver
0               Z index of 6th receiver
850             X index of 7th receiver
2               Z index of 7th receiver
900             X index of 8th receiver
2               Z index of 8th receiver
1100            X index of 9th receiver
2               Z index of 9th receiver
750             X index of 10th receiver
0               Z index of 10th receiver
0.95            Maximum damping to Cerjain's tappering
50              number of grid lines for the Cerjain's tappering
```

A.3 Mida gran

```
2299                    nx      (GRID)
2299                    nz
23000.                  lx      (DOMAIN space - time)
23000.                  lz
6.                      Tfinal
11025000000.            lambda  (Lame Parameters & density)
11025000000.            miu
2500.                   rho
500.                    alpha   (Source parameters)
0.25                    rise
950             X index of source location
0               Z index of source location
1050            X index of 1st receiver
0               Z index of 1st receiver
1150            X index of 2nd receiver
0               Z index of 2nd receiver
1250            X index of 3rd receiver
0               Z index of 3rd receiver
1350            X index of 4th receiver
0               Z index of 4th receiver
1450            X index of 5th receiver
0               Z index of 5th receiver
1550            X index of 6th receiver
0               Z index of 6th receiver
1650            X index of 7th receiver
2               Z index of 7th receiver
900             X index of 8th receiver
2               Z index of 8th receiver
1100            X index of 9th receiver
2               Z index of 9th receiver
750             X index of 10th receiver
0               Z index of 10th receiver
0.95            Maximum damping to Cerjain's tappering
50              number of grid lines for the Cerjain's tappering
```

A.4 Mida molt gran

```
4819                    nx      (GRID)
4819                    nz
55158.                  lx      (DOMAIN space - time)
22584.                  lz
17.5 Tfinal
5625000000.             lambda  (Lame Parameters & density)
5625000000.             miu
2500.                   rho
500.                    alpha   (Source parameters)
0.25                    rise
2258            X index of source location
0               Z index of source location
2378            X index of 1st receiver
0               Z index of 1st receiver
2618            X index of 2nd receiver
0               Z index of 2nd receiver
3338            X index of 3rd receiver
0               Z index of 3rd receiver
3598            X index of 4th receiver
0               Z index of 4th receiver
3658            X index of 5th receiver
0               Z index of 5th receiver
3756            X index of 6th receiver
0               Z index of 6th receiver
2258            X index of 7th receiver
10              Z index of 7th receiver
2258            X index of 8th receiver
50              Z index of 8th receiver
3658            X index of 9th receiver
10              Z index of 9th receiver
3756            X index of 10th receiver
10              Z index of 10th receiver
0.95            Maximum damping to Cerjain's tappering
50              number of grid lines for the Cerjain's tappering
```

Annex B

Taules amb mesures de temps d'execució

B.1 Original I

Servidor amb gfortran		
	Sense optimització	Nivell -O2
datain petit	405 s	316 s
datain mitjà	1418 s	1177 s
datain gran	6189 s	5118 s

Portàtil amb gfortran		
	Sense optimització	Nivell -O2
datain petit	524 s	350 s
datain mitjà	1818 s	1262 s
datain gran	7663 s	4928 s

B.2 Fusió, reordenació d'accessos i desenrollat de bucle de les derivades d'U i W

Servidor amb gfortran				
	Original	Actual	Original amb nivell -O2	Actual amb nivell -O2
datain petit	405 s	375 s	316 s	267 s
datain mitjà	1418 s	1306 s	1177 s	978 s
datain gran	6189 s	5580 s	5118 s	4200 s

Portàtil amb gfortran				
	Original	Actual	Original amb nivell -O2	Actual amb nivell -O2
datain petit	524 s	370 s	350 s	244 s
datain mitjà	1818 s	1278 s	1262 s	875 s
datain gran	7663 s	5360 s	4928 s	3735 s

B.3 Fusió, reordenació d'accessos, desenrollat de bucles de totes les derivades i matrius auxiliars globals

Servidor amb gfortran				
	Original	Actual	Original amb nivell -O2	Actual amb nivell -O2
datain petit	405 s	322 s	316 s	195 s
datain mitjà	1418 s	1120 s	1177 s	705 s
datain gran	6189 s	4708 s	5118 s	2958 s

Portàtil amb gfortran				
	Original	Actual	Original amb nivell -O2	Actual amb nivell -O2
datain petit	524 s	314 s	350 s	197 s
datain mitjà	1818 s	1082 s	1262 s	702 s
datain gran	7663 s	4524 s	4928 s	2964 s

B.4 Original II

Servidor amb gfortran		
	Sense optimització	Nivell -O2
datain petit	406 s	316 s
datain mitjà	1422 s	1178 s
datain gran	6183 s	5129 s

Portàtil amb gfortran		
	Sense optimització	Nivell -O2
datain petit	524 s	354 s
datain mitjà	1818 s	1262 s
datain gran	7668 s	5133 s

B.5 Divisió en subproblemes

Servidor amb gfortran						
	Original II	Anterior	Actual	Original II -O2	Anterior -O2	Actual -O2
datain petit	406 s	322 s	344 s	316 s	195 s	205 s
datain mitjà	1422 s	1120 s	1199 s	1178 s	705 s	734 s
datain gran	6183 s	4708 s	4803 s	5129 s	2958 s	3130 s

Portàtil amb gfortran						
	Original II	Anterior	Actual	Original II -O2	Anterior -O2	Actual -O2
datain petit	524 s	314 s	334 s	354 s	197 s	193 s
datain mitjà	1818 s	1082 s	1143 s	1262 s	702 s	694 s
datain gran	7668 s	4524 s	4796 s	5133 s	2964 s	2945 s

B.6 Original III

Servidor amb ifort				
	Original III	Actual	Original sense precisió	Actual sense precisió
datain petit	226 s	74 s	122 s	20 s
datain mitjà	826 s	261 s	446 s	68 s
datain gran	3640 s	1106 s	1941 s	273 s

Portàtil amb ifort				
	Original III	Actual	Original sense precisió	Actual sense precisió
datain petit	210 s	77 s	127 s	31 s
datain mitjà	723 s	267 s	422 s	100 s
datain gran	3148 s	1128 s	1820 s	405 s

B.7 Vectorització

Servidor amb ifort			
	Original III	Autovectorització	Vectorització a mà
datain petit	226 s	74 s	63 s
datain mitjà	826 s	261 s	224 s
datain gran	3640 s	1106 s	958 s

Servidor amb ifort, sense precisió			
	Original III	Autovectorització	Vectorització a mà
datain petit	122 s	20 s	16 s
datain mitjà	446 s	68 s	56 s
datain gran	1941 s	273 s	235 s

Portàtil amb ifort			
	Original III	Autovectorització	Vectorització a mà
datain petit	210 s	77 s	60 s
datain mitjà	723 s	267 s	212 s
datain gran	3148 s	1128 s	882 s

Portàtil amb ifort, sense precisió			
	Original III	Autovectorització	Vectorització a mà
datain petit	127 s	31 s	20 s
datain mitjà	422 s	100 s	66 s
datain gran	1820 s	405 s	261 s

B.8 POSIX Threads

Servidor amb ifort			
	Original III	Vectorització a mà	Pthreads
datain petit	226 s	63 s	49 s
datain mitjà	826 s	224 s	183 s
datain gran	3640 s	958 s	686 s

Servidor amb ifort, sense precisió			
	Original III	Vectorització a mà	Pthreads
datain petit	122 s	16 s	15 s
datain mitjà	446 s	56 s	55 s
datain gran	1941 s	235 s	289 s

Portàtil amb ifort			
	Original III	Vectorització a mà	Pthreads
datain petit	210 s	60 s	47 s
datain mitjà	723 s	212 s	167 s
datain gran	3148 s	882 s	621 s

Portàtil amb ifort, sense precisió			
	Original III	Vectorització a mà	Pthreads
datain petit	127 s	20 s	15 s
datain mitjà	422 s	66 s	49 s
datain gran	1820 s	261 s	195 s

B.9 OpenMP-sections

Servidor amb ifort			
	Original III	Vectorització a mà	OpenMP-sections
datain petit	226 s	63 s	37 s
datain mitjà	826 s	224 s	129 s
datain gran	3640 s	958 s	758 s

Servidor amb ifort, sense precisió			
	Original III	Vectorització a mà	OpenMP-sections
datain petit	122 s	16 s	12 s
datain mitjà	446 s	56 s	39 s
datain gran	1941 s	235 s	159 s

Portàtil amb ifort			
	Original III	Vectorització a mà	OpenMP-sections
datain petit	210 s	60 s	37 s
datain mitjà	723 s	212 s	120 s
datain gran	3148 s	882 s	488 s

Portàtil amb ifort, sense precisió			
	Original III	Vectorització a mà	OpenMP-sections
datain petit	127 s	20 s	12 s
datain mitjà	422 s	66 s	38 s
datain gran	1820 s	261 s	150 s

B.10 SMPSS

Servidor amb ifort			
	Original III	Vectorització a mà	SMPSS
datain petit	226 s	63 s	33 s
datain mitjà	826 s	224 s	128 s
datain gran	3640 s	958 s	383 s

Servidor amb ifort, sense precisió			
	Original III	Vectorització a mà	SMPSS
datain petit	122 s	16 s	6 s
datain mitjà	446 s	56 s	18 s
datain gran	1941 s	235 s	71 s

Portàtil amb ifort			
	Original III	Vectorització a mà	SMPSS
datain petit	210 s	60 s	35 s
datain mitjà	723 s	212 s	124 s
datain gran	3148 s	882 s	463 s

Portàtil amb ifort, sense precisió			
	Original III	Vectorització a mà	SMPSS
datain petit	127 s	20 s	12 s
datain mitjà	422 s	66 s	38 s
datain gran	1820 s	261 s	143 s

B.11 Comparativa de versions paral·lelitzades

Servidor amb ifort							
	Original III	Vectorització a mà	Pthreads	OpenMP-sect.	SMPSS	SMPSS+OpenMP	SMPSS+OpenMP solapació
datain petit	226 s	63 s	49 s	37 s	33 s	20 s	
datain mitjà	826 s	224 s	183 s	129 s	128 s	75 s	63 s
datain gran	3640 s	958 s	686 s	758 s	383 s	251 s	235 s

Servidor amb ifort, sense precisió							
	Original III	Vectorització a mà	Pthreads	OpenMP-sect.	SMPSS	SMPSS+OpenMP	SMPSS+OpenMP solapació
datain petit	122 s	16 s	15 s	12 s	6 s	6 s	
datain mitjà	446 s	56 s	55 s	39 s	18 s	16 s	13 s
datain gran	1941 s	235 s	289 s	159 s	71 s	56 s	55 s

Portàtil amb ifort						
	Original III	Vectorització a mà	Pthreads	OpenMP-sect.	SMPSS	SMPSS i solapació
datain petit	210 s	60 s	47 s	37 s	35 s	
datain mitjà	723 s	212 s	167 s	120 s	124 s	116 s
datain gran	3148 s	882 s	621 s	488 s	463 s	458 s

Portàtil amb ifort, sense precisió						
	Original III	Vectorització a mà	Pthreads	OpenMP-sect.	SMPSS	SMPSS i solapació
datain petit	127 s	20 s	15 s	12 s	12 s	
datain mitjà	422 s	66 s	49 s	38 s	38 s	35 s
datain gran	1820 s	261 s	195 s	150 s	143 s	141 s

Annex C

Informe de problemes d'autovectorització

En aquest annexe recopilem els informes de quatre versions del codi autovectoritzades. Per resumir els informes, només es mostra quines rutines **no** vectoritza el compilador d'Intel de forma automàtica.

C.1 Versió amb codi separat en subrutines

```
ifort -xSSE4.1 -vec-report2 -fp-model precise [...]
DerUW.F90(93): (col. 17) subscript too complex.
DerUW.F90(101): (col. 17) subscript too complex.
DerUW.F90(118): (col. 60) vectorization possible but seems inefficient.
DerUW.F90(137): (col. 58) vectorization possible but seems inefficient.
DerUW.F90(181): (col. 17) existence of vector dependence.
DerUW.F90(217): (col. 77) vectorization possible but seems inefficient.
DerUW.F90(245): (col. 77) vectorization possible but seems inefficient.
DerUW.F90(677): (col. 17) subscript too complex.
DerUW.F90(685): (col. 17) subscript too complex.
DerUW.F90(768): (col. 17) existence of vector dependence.
DerUW.F90(931): (col. 11) subscript too complex.
DerUW.F90(940): (col. 17) subscript too complex.
DerUW.F90(1051): (col. 14) existence of vector dependence.
DerUW.F90(1063): (col. 17) existence of vector dependence.
DerUW.F90(1352): (col. 13) subscript too complex.
DerUW.F90(1360): (col. 13) subscript too complex.
DerUW.F90(1408): (col. 13) existence of vector dependence.
psv_ssg2_homog.F90(312): (col. 9) nonstandard loop is not a vectorization candidate.
psv_ssg2_homog.F90(285): (col. 13) nonstandard loop is not a vectorization candidate.
psv_ssg2_homog.F90(299): (col. 10) nonstandard loop is not a vectorization candidate.
psv_ssg2_homog.F90(294): (col. 14) nonstandard loop is not a vectorization candidate.
psv_ssg2_homog.F90(304): (col. 17) nonstandard loop is not a vectorization candidate.
psv_ssg2_homog.F90(318): (col. 13) nonstandard loop is not a vectorization candidate.
psv_ssg2_homog.F90(332): (col. 13) nonstandard loop is not a vectorization candidate.
psv_ssg2_homog.F90(327): (col. 17) nonstandard loop is not a vectorization candidate.
psv_ssg2_homog.F90(337): (col. 17) nonstandard loop is not a vectorization candidate.
psv_ssg2_homog.F90(375): (col. 16) routine skipped: no vectorization candidates.
psv_ssg2_homog.F90(539): (col. 16) routine skipped: no vectorization candidates.
psv_ssg2_homog.F90(1410): (col. 13) existence of vector dependence.
psv_ssg2_homog.F90(1474): (col. 9) nonstandard loop is not a vectorization candidate.
```

```
psv_ssg2_homog.F90(1517): (col. 28) modifying order of operation not allowed under given switches.
psv_ssg2_homog.F90(1546): (col. 28) modifying order of operation not allowed under given switches.
psv_ssg2_homog.F90(1587): (col. 12) existence of vector dependence.
psv_ssg2_homog.F90(1597): (col. 12) existence of vector dependence.
psv_ssg2_homog.F90(1704): (col. 16) existence of vector dependence.
```

C.2 Versió amb vectors en el càlcul dels tensors

```
ifort -xSSE4.1 -vec-report2 -fp-model precise [...]
DerUW.F90(93): (col. 17) subscript too complex.
DerUW.F90(101): (col. 17) subscript too complex.
DerUW.F90(119): (col. 31) vectorization possible but seems inefficient.
DerUW.F90(138): (col. 27) vectorization possible but seems inefficient.
DerUW.F90(181): (col. 17) existence of vector dependence.
DerUW.F90(218): (col. 49) vectorization possible but seems inefficient.
DerUW.F90(246): (col. 49) vectorization possible but seems inefficient.
DerUW.F90(677): (col. 17) subscript too complex.
DerUW.F90(685): (col. 17) subscript too complex.
DerUW.F90(768): (col. 17) existence of vector dependence.
DerUW.F90(931): (col. 11) subscript too complex.
DerUW.F90(940): (col. 17) subscript too complex.
DerUW.F90(1051): (col. 14) existence of vector dependence.
DerUW.F90(1063): (col. 17) existence of vector dependence.
DerUW.F90(1352): (col. 13) subscript too complex.
DerUW.F90(1360): (col. 13) subscript too complex.
DerUW.F90(1408): (col. 13) existence of vector dependence.
DerTxxTzzTxz.F90(943): (col. 17) vectorization possible but seems inefficient.
DerTxxTzzTxz.F90(946): (col. 17) vectorization possible but seems inefficient.
DerTxxTzzTxz.F90(949): (col. 17) vectorization possible but seems inefficient.
DerTxxTzzTxz.F90(952): (col. 17) vectorization possible but seems inefficient.
DerTxxTzzTxz.F90(1108): (col. 41) subscript too complex.
DerTxxTzzTxz.F90(1404): (col. 38) subscript too complex.
DerTxxTzzTxz.F90(1406): (col. 39) subscript too complex.
DerTxxTzzTxz.F90(1408): (col. 39) subscript too complex.
DerTxxTzzTxz.F90(1410): (col. 39) subscript too complex.
DerTxxTzzTxz.F90(1430): (col. 38) subscript too complex.
DerTxxTzzTxz.F90(1432): (col. 39) subscript too complex.
DerTxxTzzTxz.F90(1434): (col. 39) subscript too complex.
DerTxxTzzTxz.F90(1436): (col. 39) subscript too complex.
DerTxxTzzTxz.F90(1456): (col. 39) subscript too complex.
DerTxxTzzTxz.F90(1458): (col. 40) subscript too complex.
DerTxxTzzTxz.F90(1460): (col. 40) subscript too complex.
DerTxxTzzTxz.F90(1462): (col. 39) subscript too complex.
DerTxxTzzTxz.F90(1486): (col. 39) subscript too complex.
DerTxxTzzTxz.F90(1472): (col. 38) subscript too complex.
DerTxxTzzTxz.F90(1479): (col. 38) subscript too complex.
psv_ssg2_homog.F90(312): (col. 9) nonstandard loop is not a vectorization candidate.
psv_ssg2_homog.F90(285): (col. 13) nonstandard loop is not a vectorization candidate.
psv_ssg2_homog.F90(299): (col. 10) nonstandard loop is not a vectorization candidate.
psv_ssg2_homog.F90(294): (col. 14) nonstandard loop is not a vectorization candidate.
psv_ssg2_homog.F90(304): (col. 17) nonstandard loop is not a vectorization candidate.
psv_ssg2_homog.F90(318): (col. 13) nonstandard loop is not a vectorization candidate.
psv_ssg2_homog.F90(332): (col. 13) nonstandard loop is not a vectorization candidate.
psv_ssg2_homog.F90(327): (col. 17) nonstandard loop is not a vectorization candidate.
psv_ssg2_homog.F90(337): (col. 17) nonstandard loop is not a vectorization candidate.
psv_ssg2_homog.F90(375): (col. 16) routine skipped: no vectorization candidates.
psv_ssg2_homog.F90(539): (col. 16) routine skipped: no vectorization candidates.
psv_ssg2_homog.F90(1410): (col. 13) existence of vector dependence.
psv_ssg2_homog.F90(1474): (col. 9) nonstandard loop is not a vectorization candidate.
psv_ssg2_homog.F90(1587): (col. 12) existence of vector dependence.
```

```
psv_ssg2_homog.F90(1597): (col. 12) existence of vector dependence.
psv_ssg2_homog.F90(1704): (col. 16) existence of vector dependence.
```

C.3 Versió amb vectorització a mà

```
DerUW.F90(25): (col. 13) subscript too complex.
DerUW.F90(33): (col. 13) subscript too complex.
DerUW.F90(53): (col. 13) existence of vector dependence.
DerUW.F90(220): (col. 13) subscript too complex.
DerUW.F90(320): (col. 13) existence of vector dependence.
DerUW.F90(839): (col. 13) subscript too complex.
DerUW.F90(901): (col. 9) vectorization possible but seems inefficient.
DerUW.F90(903): (col. 9) vectorization possible but seems inefficient.
DerUW.F90(918): (col. 13) existence of vector dependence.
DerUW.F90(1044): (col. 13) subscript too complex.
DerUW.F90(1121): (col. 9) vectorization possible but seems inefficient.
DerUW.F90(1123): (col. 9) vectorization possible but seems inefficient.
DerUW.F90(1173): (col. 13) existence of vector dependence.
DerUW.F90(1307): (col. 9) vectorization possible but seems inefficient.
DerUW.F90(1308): (col. 9) vectorization possible but seems inefficient.
DerTxxTzzTxz.F90(215): (col. 13) vectorization possible but seems inefficient.
DerTxxTzzTxz.F90(217): (col. 13) vectorization possible but seems inefficient.
DerTxxTzzTxz.F90(376): (col. 13) vectorization possible but seems inefficient.
DerTxxTzzTxz.F90(392): (col. 13) vectorization possible but seems inefficient.
DerTxxTzzTxz.F90(565): (col. 13) vectorization possible but seems inefficient.
DerTxxTzzTxz.F90(566): (col. 13) vectorization possible but seems inefficient.
DerTxxTzzTxz.F90(675): (col. 13) vectorization possible but seems inefficient.
DerTxxTzzTxz.F90(850): (col. 13) vectorization possible but seems inefficient.
DerTxxTzzTxz.F90(1075): (col. 13) vectorization possible but seems inefficient.
psv_ssg2_homog.F90(871): (col. 12) vectorization possible but seems inefficient.
psv_ssg2_homog.F90(1095): (col. 12) vectorization possible but seems inefficient.
psv_ssg2_homog.F90(1096): (col. 12) vectorization possible but seems inefficient.
psv_ssg2_homog.F90(312): (col. 16) routine skipped: no vectorization candidates.
psv_ssg2_homog.F90(473): (col. 16) routine skipped: no vectorization candidates.
psv_ssg2_homog.F90(1326): (col. 13) existence of vector dependence.
psv_ssg2_homog.F90(1390): (col. 9) nonstandard loop is not a vectorization candidate.
psv_ssg2_homog.F90(1433): (col. 28) not allowed under given switches.
psv_ssg2_homog.F90(1462): (col. 28) not allowed under given switches.
psv_ssg2_homog.F90(1503): (col. 12) existence of vector dependence.
psv_ssg2_homog.F90(1513): (col. 12) existence of vector dependence.
psv_ssg2_homog.F90(1620): (col. 16) existence of vector dependence.
```

Annex D

Glossari

- **API (Application Programming Interface), interfíce**: conjunt de funcions ja dissenyades i ben documentades que tenen com objectiu facilitar una tasca concreta al programador.
- **Directiva**: instrucció inclosa en un programa que interpreta el compilador, sovint efectuant una fase previa a la compilació anomenada preprocesament, amb una finalitat específica.
- **Graf Aciclic Dirigit**: estructura que estableix relacions entre parelles d'elements. És dirigit perquè les relacions són unidireccionals. És acíclic perquè partint d'un element qualsevol, i seguint un camí de relacions, no tornem a l'element de partida en cap moment.
- **OpenMP**: interfície (veure més amunt) per paral·lelitzar programes, ampliament implementada.
- **Optimitzar**: modificar un algorisme perquè aprofiti millor els recursos del sistema, o millori el seu rendiment i s'executi en menys temps.
- **Overhead**: excès de feina, d'ús de recursos o de computació indirecta, necessària per assolir un fi relacionat amb els computadors.
- **Paral·lelitzar**: modificar un algorisme, dividint-lo en parts que poden ser executades per diferents nuclis o processadors.
- **Profiling**: anàlisi d'un programa a partir de les dades obtingudes mitjançant mesures mentre aquest s'executa.
- **Pthreads o POSIX Threads**: una altra interfície per paral·lelitzar, estandarditzada.
- **SMPSS**: interfície dissenyada a partir d'una altra interfície, CellSS, que busca automatitzar el paral·lelisme en arquitectures amb múltiples processadors.
- **Stencil, o plantilla**: operació que consisteix en obtenir un valor concret a partir de valors relacionats i situats d'una forma determinada en una estructura.
- **Thread, o fil**: component d'execució en la qual es pot dividir un programa. Un fil és una instància d'execució d'un programa: comparteix codi i recursos amb aquest programa, i executa una part o la totalitat del codi d'aquest.

- **Tile, rajola o submatriu**: subconjunt dels elements d'una matriu, que també té forma rectangular.
- **Variable d'entorn**: valors dinàmics associats normalment a un sistema operatiu, que poden afectar a la forma d'execució dels processos en el computador.

Bibliografia

[1] Rojas, Otilio J. *Mimetic Finite Difference Modeling Of 2D Elastic P-SV Wave Propagation*. Computational Science Research Center, San Diego, CA. 2007.

[2] Saad, Yousef. *Iterative methods for sparse linear systems, Second Edition*. Society for Industrial and Applied Mathematics. 2003.

[3] D. Jiménez, E. Morancho, À. Ramírez *Programació Conscient de l'Arquitectura. Tema 1: Introducció*. Departament d'Arquitectura de Computadors. Universitat Politècnica de Catalunya. 2008.

[4] D. Jiménez, E. Morancho, À. Ramírez *Programació Conscient de l'Arquitectura. Tema 2: Eines*. Departament d'Arquitectura de Computadors. Universitat Politècnica de Catalunya. 2008.

[5] D. Jiménez, E. Morancho, À. Ramírez *Programació Conscient de l'Arquitectura. Tema 6: Instruccions vectorials*. Departament d'Arquitectura de Computadors. Universitat Politècnica de Catalunya. 2008.

[6] Alejandro Duran, Josep M. Perez, Eduard Ayguadé, Rosa M. Badia, Jesús Labarta. *Extending the OpenMP Tasking Model to Allow Dependent Tasks*. Barcelona Supercomputing Center. Universitat Politècnica de Catalunya. 2008.

[7] Eduard Ayguadé, Bob Blainey, Alejandro Duran, Jesús Labarta, Francisco Martínez, Xavier Martorell, Raúl Silvera. *Is the 'schedule' clause really necessary in OpenMP?*. CEPBA-IBM Research Institute, Departament d'Arquitectura de Computadors Universitat Politècnica de Catalunya / IBM Toronto Lab. 2003.

[8] Jiayuan Meng, Kevin Skadron. *Performance Modelling and Automatic Ghost Zone Optimitzation for Iterative Stencil Loops on GPUs*. Departament of Computer Science. University of Virginia. 2009.
Una còpia es pot trobar a `www.cs.virginia.edu/~skadron/Papers/meng_stencil_ics09.pdf`

[9] Robert Strzodka, Mohammed Shaheen, Dawid Pajak, Hans-Peter Seidel. *Cache Oblivious Parallelograms in Iterative Stencil Computations*. ACM. 2010.

[10] Dolors Costa, M. Ribera Sancho, Ernest Teniente. *Enginyeria del software: Especificació*. Edicions UPC. 2000.

[11] Tanja van Mourik. *Fortran 90/95 Programming Manual, fifth revision*. Chemistry Departmen. University College London. 2005

[12] *SMP Superscalar (SMPSS) User's Manual*. Version 2.1. Barcelona Supercomputing Center. 2009

[13] *Intel Fortran Programmer's Reference*. Intel Corporation. 2002.

[14] *Intel C++ Compiler for Linux Intrinsics Reference*. Intel Corporation. 2006.

[15] *Optimizing Applications with Intel C++ and Fortran Compilers for Windows, Linux and Mac OS X* Version 11.x. Intel Corporation. 2009.

[16] *OpenMP Application Program Interface*. Version 3.0. OpenMP Architecture Review Board. 2008.

[17] Amdahl's Law. `http://en.wikipedia.org/wiki/Amdahl's_law`

[18] Seismic Waves. `http://en.wikipedia.org/wiki/Seismic_wave`